Sweet-expressions: Version 0.2

dwheeler.com (https://www.dwheeler.com/readable/version02.html)

by David A. Wheeler (http://www.dwheeler.com), 2007-01-05 revised 2010-11-12

This page is obsolete; see http://readable.sourceforge.net (http://readable.sourceforge.net) instead.

This page shows the proposal for version 0.2 of sweet-expressions, and shows that sweet-expressions are very general by showing what they look like in many different code fractions in many Lisp-based languages: Scheme, Common Lisp, Arc, ACL2, PVS, BitC, AutoCAD Lisp (AutoLisp), Emacs Lisp, SUO-KIF, Scheme Shell (Scsh), GCC Register Transfer Language (RTL), MiddleEndLispTranslator (MELT), Satisfiability Modulo Theories Library (SMT-LIB), NewLisp, Clojure, and ISLisp.

Various people have had a chance to look over version 0.1 of sweet-expressions. Many people liked the idea, but had lots of useful feedback. In addition, I've learned things based on experimenting with a prototype implementation of version 0.1. So, here's version 0.2 of sweet-expressions.

I still call these “sweet-expressions”, because by adding syntactic sugar (which are essentially abbreviations), I hope to create a sweeter result. Sweet-expressions are still implemented as a change to the *reader*, thus, ***Lisp macros can work as-is***.

I devised two variations of sweet-expressions, "infix default" and "infix not default", and then tried out many different code fragments to see which one seemed to work better. I expected "infix default" to be better, but after experimentation with real code, it appears that their additional complexity wasn't

really worth it. I found that surprising, but that's why experiments are so valuable. The much simpler "infix non-default" alternative appears to be nearly as easy to read (in general), and its rules are much simpler.

I experiment with a variety of real-world Lisps, because *any new Lisp syntax must be independent of the underlying semantics*. M-expressions, ACL2 infix, and many other "more readable Lisp syntax" efforts failed because they weren't independent of the underlying semantics. Now that we know *why* they failed, we can avoid their mistakes, and by using many real-world Lisp dialects, I can make sure that they're useful too.

Quick Examples

Let's do two quick examples - we'll use sweet-expressions to represent calculating factorials and Fibonacci numbers, in both cases using Scheme:

(Ugly) S-expression Sweet-expression 0.2

```
(define (fibfast n)
  (if (< n 2)
    n
    (fibup n 2 1 0)))
```

```
define fibfast(n)  ; Typical function notation
  if {n < 2}       ; Indentation, infix {...}
    n              ; Single expr = no new list
    fibup(n 2 1 0) ; Simple function calls
```

```
(define (fibup max count n-1 n-2)
  (if (= max count)
    (+ n-1 n-2)
    (fibup max (+ count 1) (+ n-1 n-2) n-1)))
```

```
define fibup(max count n-1 n-2)
  if {max = count}
    {n-1 + n-2}
    fibup(max {count + 1} {n-1 + n-2} n-1)
```

```
(define (factorial n)
  (if (<= n 1)
    1
    (* n (factorial (- n 1)))))
```

```
define factorial(n)
  if {n <= 1}
    1
    {n * factorial{n - 1}} ; f{...} => f({...})
```

Note that you can use traditional math notation for functions; `fibfast(n)` maps to `(fibfast n)`. Infix processing is marked with {...}; `{n <= 2}` maps to `(<= n 2)`. Indentation is significant, unless disabled by (...), [...], or {...}. This example uses variable names with embedded "-" characters; that's not a problem, because the infix operators must be surrounded by whitespace and are only used when {...} requests them.

It's actually quite common to have a function call pass one parameter, where the parameter is calculated using infix notation. Thus, there's a rule to simplify this common case (the prefix {} rule). So factorial{n - 1} maps to factorial({n - 1}) which maps to (factorial (- n 1)).

Credit where credit is due: The Fibonacci number code is loosely based on an example by Hanson Char (http://hansonchar.blogspot.com/2006/01/fibonacci-numbers-in-scheme.html).

General Sweet-expression version 0.2 rules

Sweet-expressions are simply extensions to traditional s-expressions, and are implemented by adding capabilities to the reader (usually named "read"). A sweet-expression reader can read ordinary s-expressions as usual, but can also read other syntax that are handy abbreviations - just as 'a is a handy abbreviation of (quote a). Thus, macros and other complex Lisp capabilities work as-is, without change.

Here are the draft sweet-expressions version 0.2 rules:

1. **Indentation**. Indentation is meaningful; the "I-expressions" of Scheme SRFI-49 (http://srfi.schemers.org/srfi-49/srfi-49.html) are supported, with the change that blank lines at the beginning of a new expression are ignored. Thus an indented line is a parameter of its parent, later terms on a line are parameters of the first term, and lists of lists are marked with "group". A line with exactly one datum, and no child lines, is simply that item; otherwise that line and its child lines are themselves a new list. Top-level statements' indentation is ignored; they *should* begin at the left edge. Indentation is disabled inside the grouping pairs (), [], and {}, whether they are prefixed or not. A comment-only line is always completely ignored (this is a line that begins with zero or more tabs/spaces, followed by ";"). ▸ *A blank line always terminates a datum, so once you've entered a complete expression, "Enter Enter" will always end it. The "blank lines at the beginning are ignored" rule eliminates a usability problem with the original I-expression spec, in which two sequential blank lines surprisingly return (). A function call with 0 parameters must be surrounded or immediately followed by a pair of parentheses: (pi) or pi(). Indentation can be optionally disabled, to improve backwards-compatibility and simplify command line interaction when such interactions are small; this variation is called sweet-expressions-noindent.*
2. **Prefixed (...)**. Syntax of the form **e(...)** — with **no whitespace** between symbol or list e and the open parenthesis — are mapped to **(e ...)**. Any parameters in "..." are space-separated. This produces another expression, so

this can be repeated (left-to-right). ‣ *This adds support for traditional function notation. For example, "cos(x)" maps to "(cos x)", "max(3 4)" maps to "(max 3 4)", and "f(x)(a b)" maps to "((f x) a b)". Note that this is especially convenient for certain styles of functional programming, including lambda expressions; lambda((x) {x + x})(4) when executed returns 8.*

3. **Unprefixed (...)**. (...) disable indentation processing inside them. A common extension must be supported: (. x) must mean x. Implementations may have a special mode where an expression beginning with an unprefixed open parenthesis "(" begins a traditional s-expression until its matching ")", but since it is hard to use sweet-expressions this way, this would not normally be the default. ‣ *If an implementations called a "standard" s-expression reader, it would be extremely backward-compatible with essentially all existing Lisp files. However, this mode is hard to use... it would mean that you must use [...] for lists, and failure to do so would produce mysterious errors. The (. x) rule is a common extension in Scheme implementations; it's required here to be consistent and provide a simple way to escape I-expression's "group" term (an alternative is "group group"). Note that any "(" preceded by whitespace, "(", "{", or "[" is unprefixed.*

4. **Unprefixed {...}, aka infix**. An unprefixed {...} contains an "infix list". If the enclosed infix list has (1) an odd number of parameters, (2) at least 3 parameters, and (3) all even parameters are the same symbol, then it is mapped to "(*even-parameter odd-parameters*)". Otherwise, it is mapped to "(nfx *list*)" — you'll need to have a macro named "nfx" to use it. ‣ *This rule means that {n = 0} maps to (= n 0), {3 + 4 + 5} maps to (+ 3 4 5), {3 + {4 * 5}} maps to (+ 3 (* 4 5)), and {3 + 4 * 5} maps to (nfx 3 + 4 * 5). Use prefixed parentheses for prefix functions, e.g., {-(x) * y} maps to (* (- x) y). Consistently using {...} so infix operators are always equal in a particular list has the advantage that **all** macros will see the usual list form - with the function in the first position. If you want operator precedence, define an nfx macro to implement the precedence rules you desire. Or, if you **never** want precedence, define nfx to be an error. The even parameters must be exactly the same symbol; pointer equality such as Scheme's eq? is a good way to test*

this. Every infix operator must be surrounded by whitespace for this rule to work as designed.

5. **Prefixed {...}**. A prefixed expression f{...}, where f is a symbol or list, is an abbreviation for f({...}). ‣ *This rule simplifies combining function calls and infix expressions when there is only one parameter to the function call. This is a common case; for example, "not" (which normally is given only one parameter) often encloses infix "and" and "or". Thus, f{n - 1} maps to (f (- n 1)). When there is more than one function parameter, use the normal term-prefixing format instead, e.g., f({x - 1} {y - 1}) maps to (f (- x 1) (- y 1)).*

6. **Unprefixed [...]**. Unprefixed square brackets [..] disable indentation processing inside them. ‣ *Use [...], instead of (...), to get the benefit of a list without disabling sweet-expression capabilities, even if you've turned on a mode where (...) surround traditional s-expressions. The contents are not interpreted as infix, so [a + b(x)] maps to (a + (b x)).*

7. **Prefixed [...]**. Prefixed square brackets e[...], where e is a symbol or list, maps to (bracketaccess e ...). ‣ *Thus, "t[x]" maps to "(bracketaccess t x)". This is intended to simplify use of indexed arrays, associative arrays, and similar constructs. You could even define bracketaccess as a macro that simply returns its arguments; in this case f[5] would eventually map to (f 5).* (Still under discussion: Should t[x] mean the same as t(x)?)

Note that usual Lisp quoting rules still work, so 'a still maps to (quote a). But they work with the new capabilities, so 'f(x) maps to (quote (f x)). Same with quasiquoting and comma-lifting. A ";" still begins a comment that continues to the end of a line, and "#" still begins special processing.

Implementations are likely to call underlying implementations when they encounter "#", so don't expect that expressions beginning with "#" will continue to suport sweet-expressions. For example, in Scheme, use vector(...) instead of #(...). *Many Scheme implementations have nonstandard extensions for "#", so a portable sweet-*

reader can't easily reimplement the functionality of a local "#". Nor can the sweet-reader easily call on the underlying implementation of "#"; because Scheme only supports a one-character peek with no unget character.

Typical implementations would have a function "sweet-read" that implements the above (with indentation meaningful), and a "sweet-read-noindent" that implements the above when indentation in not meaningful. Sweet-read-noindent has some advantages in some circumstances: (1) simple command line interaction (because you don't need to type "Enter" twice to execute), and (2) when compatibility with existing files is critical (because indentation is also ignored in traditional S-expressions).

There should probably be an optional parameter to change how unprefixed (...) is interpreted:

- It could be forbidden (other than perhaps simple cases like enclosing zero or more symbols). This coding style makes it easier to distinguish code intended for a traditional s-expression reader, and prevents misinterpretion if whitespace is accidentally inserted in front of an open parenthesis.
- It could also be interpreted as being the same as unprefixed [...].

Note that changing the interpretation of unprefixed (...) would mean loss of compatibility with existing files. For new languages, or where the reader would only be expected to read sweet-expression files, that's probably irrelevant.

Other potential options could forbid allowing top-level statements to be indented (as Python does), and change prefixed [] to be interpreted the same way as prefixed (). These options could be useful in certain cases.

Future editions may add a when indentation is meaningful, that is, not inside (...), [...], or {...}.

Comments and Implications

Here are a few comments about the rules, particularly about their implications.

Note that you *have* to disable indentation to use infix operators as infix operators; everything inside {...} has indentation ignored. This doesn't seem to be a problem in practice, in fact, it seems to be a slight advantage.

With sweet-expressions, you can use the traditional Lisp read-eval-print loop as a calculator, as long as you remember to surround infix expressions with {...} and surround infix operators with whitespace. For example, "{3 + 4}" will be mapped to (+ 3 4), which when executed will produce "7". Use normal function notation for unary functions, e.g., "{-(x) / 2}" maps to "(/ (- x) 2)". Nest {...} when you need to, e.g., "{3 + {4 * 5}}" will map to "(+ 3 (* 4 5))". If you mix infix operators at the same level, you must have an "nfx" macro defined to handle precedence, and you must be careful about other macros you use.

Notice that since all the transforms happen in the *reader*, sweet-expressions are highly compatible with macros. Sweet-expressions simply define new abbreviations, just as 'x became (over time) a standard abbreviation for (quote x). As long as simple infix expressions are used (ones that don't create nfx), after reading the expressions *all* expressions are normal s-expressions, with the operator at the initial position. So macros defined by Common Lisp's macros, etc., will work as expected. Common Lisp has some hideously confusing terminology, though. Common Lisp has macros, but it also has a completely different capability: "macro characters", which introduce "reader macros" - i.e., hooks into the reader used during read time. The Common Lisp Hyperspec clearly states in its glossary on macro characters, "macro characters have nothing to do with macros", but I think they should have chosen a name that had nothing to do with macros as well. Obviously sweet-expressions *can* affect macro characters, since they implement a

different reading syntax. This doesn't affect most real Common Lisp programs, which often avoid macro characters anyway. Common Lisp macro functions (e.g., defmacro and macrolet) work just fine with sweet-expressions.

A sweet-expression reader can read most Lisp files as-is. In the rare cases where this isn't true, use a pretty-printer on the original Lisp file - essentially any pretty-printer will fix the problems. There are two main reasons a Lisp file can't be read using a sweet-expression reader:

- A later top-level expression begins with an indented line, yet it has no blank lines preceding it. The problem is that such a line looks like a child line. You can fix this by making the indentation start on the left edge, or by inserting a blank line between them.
- A line contains more than one top-level s-expression. For example, the following would be read by a sweet-reader as-is:

  ```
  (define a (x) (b) cd e (f (g h)))
  ```

  But lines that have more than one top-level expression in the same line will be interpreted differently, because the later expressions will be interpreted as arguments to the first one:

  ```
  (define x 10) (define y 20)
  ```

  You can fix this by breaking such lines into multiple lines with the same indentation level.

A sweet-expression pretty-printer would need to determine which functions should be used in infix expressions. I recommend that in Scheme, the sequence => should *not* be considered an infix operator. Otherwise, I recommend that the following default to being infix operators:

- A sequence of 1-5 punctuation characters, namely, the Unicode math characters (U+2200 through U+22FF) and this set: + - / * < > = & |
- One to five colons (:). *The colon is often a namespace manipulator, as well an indicator of parameter keywords, so combining it with other characters can be problematic.*
- Whatever is mapped by ||. *This may be the "null" atom.*
- The words "and", "or", and "xor", in upper or lower or initial-upper case. *The terms "and" and "or" are special forms in essentially all Lisps, since they typically short-circuit. Function xor essentially always means "exclusive-or" when used, so for consistency it makes sense to treat it the same way as "and" and "or".*

In circumstances where there is no previous code to be compatible with (and possibly no existing s-expression reader), a reader might choose to implement unprefixed (...) as unprefixed [...]. After all, [...] has nearly the same meaning. Implementing (...) as {...} is probably a bad idea; it would probably be surprising in many cases, and would interfere with the mental model that "you must use {} to use infix ordering".

This version of sweet-expressions uses unprefixed {} to mark infix expressions. This is highly compatible with other Lisps, and the "Unprefixed {}" rule would be a great backwards-compatible addition to the *standard* reader of Scheme and Common Lisp. Scheme specifically reserves {...} for future use (R5RS section 2.3, R6RS section 4.21). Common Lisp does not define {} (see section 2.4 of the Common Lisp Hyperspec, based on ANSI Common Lisp X3.226). BitC spec version 0.10 (June 17, 2006) section 2.4.3 also reserves {...}. Similarly, this use of [] is not a shocking incompatibility. Common Lisp and Scheme R5RS reserve [] for future use, and Scheme R6RS (section 4.3) uses [...] as an alternative symbol pair for (...), very similar to its meaning in sweet-expressions version 0.2. I'd love to see the "unprefixed {}" rule implemented as a common built-in extension in standard readers; in Common Lisp it'd be an especially trivial addition to the readtable.

In an earlier draft I generated a macro named "infix-fix" when precedence processing was required, but this requires an embedded "-". Some variants may want to "automatically" separate these, so that x*y is converted into (* x y), and typically infix macros are named "nfx" anyway (to simplify direct use). So I switched to what appears to be the most common convention, "nfx".

I'm well aware that there are some who don't like any change in Lisp notation. Some of these people seem to believe that the current Lisp notation was handed down from on high, never to be changed. Well, you don't have to use improvements like this, or even agree that they are improvements. But most developers have abandoned Lisp precisely because of Lisp's hideous, inadequate notation (and I say that as someone who has used Lisp for decades). Lisp notation was *not* handed down from on high, and it *has* changed over time. The "LISP 1.5 Programmer's Manual" (by John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart and Michael I. Levin; The M.I.T. Press, 1962, second edition) (http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf/view) describes the parent of all modern Lisp-based systems. (Note that even LISP's creator didn't think much of using S-expressions as a programming notation.) LISP 1.5 did *not* have a ' operator - you had to say (QUOTE X). It didn't have abbreviations for quasiquoting (`) or comma-lifting (,) either. Today, people would not accept a Lisp that didn't at least have the common abbreviation for QUOTE. Indeed, Tony Hasemar's book "A Beginner's Guide to Lisp" (1984) says in the second page of the Foreward, "do NOT buy a Lisp which does not allow the single-quote sign in place of the word QUOTE, unless you have absolutely no alternative". Lisp notation has been stagnant for a while; it's time to add modern conveniences as abbreviations.

Some objections don't seem to realize that this proposal is different. It's true that there have been many abandoned efforts of the past to improve on S-expressions, but I think all those efforts failed to realize that any replacement for S-expressions

must be *completely general*, just as S-expressions are, and *not* tied to a particular semantic. Practically all past efforts, such as M-expressions and similar work, failed precisely because they weren't general enough. It's true that tooling support is necessary for any notation like this (e.g., in program editors), but that's why a *standard* format needs to be defined so tools can implement it (and not 1000 application-unique reader macros). There's no reason tools can't support sweet-expressions as well as they support s-expressions today.

I think most software developers will ***not*** agreeably use a Lisp-based language unless that language has better built-in support for an easy-to-read programming notation. Programs must be read by others, and if the programming notation is odious to read, then the language has a key flaw. Most developers think Lisp is odious to read, even *after* they've used it for a while. If the Lisps won't provide an easy-to-read notation, those developers will just use another language that's more user-friendly (even when it's less appropriate for their problem) - *and that is precisely what they are doing*. Here, we try to learn from the past, keep all of S-expression's benefits, but provide a better notation that *others can read*.

What's Changed?

Here are the improvements of sweet-expressions over version 0.1:

1. This version is *much* more compatible with existing Lisp code, and the rules are simpler. In the previous version, infix was automatically detected, but this required function name patterns that had to be memorized as well as an "escape" mechanism. Now each infix list is specially marked using {...}; since this marking isn't part of traditional Lisp syntax, there's no risk of misinterpretation. There's no need to have special special-case detection of f(...) where ... is an infix expression; instead, f{} explicitly states that ... is an infix expression (this kind of explicitness reduces some risks of

misinterpretation). In short, the auto-detection rules became complicated (to be useful), and required additional "escaping" rules for additional complications, without enough benefit. It's true that {...} can look like (...), but [...] is normally used for non-infix lists while *many* languages use {...} for block structures, so this appeared to be the best option.
2. It can work more cleanly with macros that provide infix precedence (for those who want precedence rules).
3. It extends version 0.1's "name-prefixing" into "expression-prefixing". This is not only more general, it also makes certain kinds of functional programming much more pleasant.
4. It adds syntax for the common case of accessing maps (such as indexed or associative arrays) - now a[j] is translated into (bracketaccess a j).

Infix: Default or not?

I originally considered two alternatives: infix as the default (infix operators were automatically inferred from their position and pattern), and infix is not the default (you have to mark every list using infix rules with {...}). I tried to create the best "infix is the default" rules that I could, did the same with "infix is not the default" rules, and then tried them out on many program fragments to compare them. I expected "infix as default" to win, but surprisingly, "infix is not default" turned out to be convincingly better. No one was surprised more than me.

Arguments for and against infix default

There were arguments for both infix as default, and for infix as not the default:

- Arguments for "infix default": I thought that programs written using the "infix default" text would tend to look a little cleaner if they use infix

operators often. There's also a good argument based on having "reasonable defaults": most of the world expects infix operators to be used as infix operators, so having them *default* to being infix operators makes sense.
- Arguments for "infix non-default": The rules for infix non-default are *much* simpler, both to explain and to program (including other tools, etc.). It could be argued that they're easier to reason about, too, just because they are so much simpler. This means that it might be easier to get broader buy-in for the "infix non-default" option.

I had expected "infix default" to be the eventual winner. My first version of sweet-expressions (version 0.1) uses "infix default", and I argued strongly for good defaults on the mailing list. But when I extracted a number of sample programs, and tried out both alternatives, the "infix non-default" versions were very similar to the "infix default" ones (both in readability and length). This strongly suggests that all the extra rules for automatic detection of infix weren't worth the significant complexity in the rules. So in this current draft, I show the "infix is not default" version of the rules as the draft rules.

There seem to be several reasons that "automatic infix" doesn't actually get USED very often in real code, now that I examine the code fragments:

- When infix operators are used as conditionals (e.g., in "if" and "cond"), it seems very natural to put the conditional on the same line as other terms (e.g., the "if" or the result of the "cond"). In such cases, you need to surround the infix expression with something like {...} anyway.
- I believe that a *reader* should avoid preset precedence, at least in most cases (see the previous paper for an explanation of this). Thus, if you also want to avoid invoking a macro, you need to surround subexpressions with {} when you mix operators, e.g., {3 + {4 * 5}}. This style means that every resulting list is clearly marked syntactically.

- When you're sending operators in a function call, if you use f(...), then with multiple parameters you need to surround the various infix operators somehow anyway... e.g., f({1 + 2} {3 + 4}). The one special case is the one-parameter call using infix, but prefixed {} such as "f{n + 3}" handles that case nicely.

I *had* to use "infix default" rules in version 0.1, because I only used parentheses for grouping and thus didn't have reasonable ways of marking distinctions. But once unprefixed "(...)" became "s-expressions, as-is", and I allowed the use of [...] and {...} for other purposes, other trade-offs changed.

Infix default rules

The "infix default" rule alternative replaced the "Unprefixed {...}, aka infix" and "Prefixed {...}" rules with these rules:

1. **Infix Detection**. Expressions are automatically interpreted as infix if they (1) have an odd number of parameters of at least 3, (2) all even parameters are infix operators, and (3) the first parameter is *not* an infix operator. You must separate each infix operator with whitespace on both sides. Otherwise, expressions are interpreted as normal “function first” prefix notation. Infix expressions must have an odd number of parameters, so 3 + 3 + 2 is fine, but 3 + 3 + is not. Use the "name-prefixed" form for unary operations, e.g., -(x) for "negate x". Note that you do not have to specially mark infix expressions; they are automatically determined.
2. **Infix operators**. In Scheme, the sequence => is *never* an infix operator. Otherwise, infix operators are:
    - A sequence of 1-5 punctuation characters from the Unicode mathematical characters (U+2200 through U+22FF) and this set: + - / * < > = & |

- One or more colons (:)
- Whatever is mapped by || (which may be the "null" atom)
- The words "and" and "or", in upper or lower or mixed case. (These are special forms in essentially all Lisps, since they short-circuit.)
- An atom beginning and ending with "/", enclosing a valid non-null function name. The function name can't have whitespace, it can't have grouping characters like parentheses, and it can't begin with a digit. The name of the infix function is the set of one or more characters so enclosed. Thus, {a /convoke/ b} is an infix expression, translating to (convoke a b). This is a lower-precedence rule than the ones above, so /+/ is the 3-character operator name "/+/", not the one-character operator name "+".

3. **Curly braces and Translation**. In an infix expression, if there's only one even parameter, or all the even parameters are equal, it is translated to the list "(*even-parameter list-of-odd-parameters*)". Otherwise, it is translated to "(nfx *list*)". Thus, {n = 0} maps to (= n 0), {3 + 4 + 5} maps to (+ 3 4 5), and {3 + 4 * 5} maps to (nfx 3 + 4 * 5). This means that if you have an nfx macro that support precedence, you can use precedence, but macros that operate on lists *before* the nfx macros executes will see the unmodified list. The {...} braces create new lists that may have infix inside; Use them to group lists using different infix operators if you wish to avoid invoking an nfx macro. Thus, {3 + {4 * 5}} will be translated by the reader into (+ 3 (* 4 5)), without invoking nfx. Consistently using {...} so infix operators are never mixed in a list has the advantage that *all* macros will see the usual list form. But if you want precedence, you can define nfx to provide what you need.

4. **Prefixed {}**. Prefixed {} is interpreted identically to prefixed (), so f{a} is equivalent to f(a). This is to satisfy those who just want to use {} everywhere.

5. **Disabling Infix**. To disable infix interpretation, use unprefixed [...], unprefixed (...), or surround the second parameter with as(...).

Here are some rule ramifications and tweaks for the infix-default case:

1. In "Prefixed ()", If infix is default, then if the content of the parens can be considered an infix expression, it is considered as an infix expression of one parameter - so if infix is default, f(2 + 3) maps to (f (+ 2 3)).
2. In immediate execution, If infix is default, then if the line can be interpreted as a complete infix expression (see below), it is interpreted that way.
3. In "Unprefixed []", if infix is default, they also disable infix interpretation of their immediate contents.

In circumstances where there is no previous code to be compatible with (and possibly no existing s-expression reader), a reader might choose to implement (...) as {...}. That way, code can look *very* similar to "traditional" notations.

One advantage of "infix default" is that, on the command line, you can type 3 + 4 starting at the left edge and press "Enter" after entering the expression. This will be immediately transformed into (+ 3 4), and then immediately executed - producing 7, one hopes. That's a very nice side-effect, but as the programs got larger, this advantage became less obvious.

Indentation

There are some hidden complications in supporting indentation with Lisp-like languages. This section talks about them in general, and then discusses "immediate completion".

Indentation Issues

Here's an example of an important issue. What should you do when you read in code that is indented at the top level like this?:

```
    x
    y
    z
```

One interpretation is that there should be 3 different results: x, y, and z. But consider how this would be read. You'd read in the indentation before x, and note that as the "topmost" indentation. Then you'd read in the indentation before y, notice that it was the same as x's, and stop just before reading the "y" and return with just "x". But wait - if you did that, when you read "y" you would think that there was *no* indentation (the previous read consumed it), and thus z would be further indented... returning (y z). Ooops, that can't be right.

Since essentially the dawn of Lisp in the 1950s there has been a "read" function that reads an S-expression from the input and returns it. This is an extremely stable function interface, and one not easily changed in fundamental ways. In particular, no user of "read" expects it to *also* return some state - such as the indentation that was read the *last* time read was called - and certainly they aren't going to provide that information back to "read" anyway. Not only is this difficult to change for backwards-compatibility reasons, it's not clear you should - simple interfaces are a good idea, if you can get them, and adding such "indentation state" as a required parameter would certainly complicate the interface.

In theory, you could "unget" all the indentation characters, so that the next read would work correctly. But the support for this is rare; for example, Scheme doesn't even *have* a standard unget character function, and the Common Lisp standard only supports one character unget (not enough!).

You could store "hidden state" inside the read function. Problem is, character-reading is not the exclusive domain of the read function; many other functions read characters, and they are unlikely to look at this hidden state. These functions tend to be low-level functions and in some implementations are difficult to override. What's more, you would have to store hidden state for each possible input source, and this can become insane in the many implementations that support support ports of non-files (such as from strings). "Hidden state" could allow for all this, but the hideous complications of *implementing* hidden state suggests that it'd be better to spec something that does *not* require hidden state.

We could *require* that the top-level line begin at the left edge. This is not unknown; Python, a popular language using indentation, *requires* that the top level begin at the left edge (and raises an error if an attempt is made otherwise). This completely eliminates the need for hidden state - top level statements only start at the left edge, so there's nothing to remember.

For backwards-compatibility, we can make a slight concession: *allow* the top level to be indented, but *completely ignore* its indentation. It would be wise for new code to begin at the left edge, because otherwise the code could be misleading (this would be easy to check). But sometimes older files are indented and begin immediately with "("; in such cases, as long as there are blank lines between the top-level statements, they could be left as-is. The example above then becomes misleading; it will result in (x y z), because the indentation of "x" is ignored, making y and z indented under x. To be fair, most programs aren't formatted this way; as long as there are blank lines between top-level statements, the indentation of the top line is irrelevant (and even when they're not separated, they'll still often be interpreted correctly). Where backwards-compatibility is less important than countering the risk of misleading indentation, we could optionally forbid indented top-level statements (as Python does).

The demo implementation of I-expressions (as defined in SRFI-49) actually ignores indentation of a top-level statement. However, nowhere the does spec clearly *document* this, nor explain why this occurs. So, it's documented here.

Immediate completion

Originally in sweet-expressions there was this additional rule:

> 1. **Immediate completion**. A single complete expression that begins at the left edge of a line, and is *immediately* followed by newline after it completes, causes the expression to immediately complete (without waiting for the next line of input). ‣ *Thus, entering* `load("hello")` *and immediately pressing return will execute immediately, but* `load "hello"` *will require pressing Enter twice (because there might be another line to follow). This rule makes interactive use more pleasant, without harming file reading. It avoids two problems: (1) having to press Enter twice to execute simple one-line commands, and (2) allowing the command line to easily go "out of sync" (when, after pressing return, the user sees the result of the* ***previous*** *line). Type an initial space if you want to enter only a single complete expression on a line yet follow it with child lines.*

However, it turns out that there are awkward interactions between this and the indentation rule. The basic problem is that, unless the reader maintains per-port state information, when it's called again it has no way to know what the "current" top-level indentation is. This is because the initial whitespace of a line may have already been consumed by a *previous* call to read. If the first line of a previous expression began 3 spaces in, you'll need to examine those 3 spaces and then the next character before you can determine if the current line is a child of the first expression, or the start of a new one. And once read, you can't easily return those

characters in most Lisp systems. Scheme doesn't have an unget character function at all, and Common Lisp's is limited to only one character, and "peek" will only give one-character lookahead... so simple solutions to "undo" this reading don't work.

As a result, you often have no way to *know* if you're really at the "left edge" unless hidden per-port state information is hidden away by read. Maintaining per-port state information is a serious problem for many implementations (especially since the input may be from a calculated string!), and it's certain to interfere with other reading functions. As a result, it's better to not spec something that would inadvertantly require maintaining this hidden state.

Indeed, this has ramifications for any indentation processing. Basically, you should start topmost expressions at the leftmost edge, or include a blank line between expressions, because of this.

Code Samples

Here are a number of examples, which show the impact of the rules, and led me to believe that "infix non-default" was actually the better option. Thus, "infix non-default" examples use the rules for draft version 0.2, above; the "infix default" examples show the impact of that alternative. Since I am only using code snippets, as examples for research and commentary, I claim that these snippets are "fair use" under copyright law. I *do* try to acknowledge my sources (when I know them).

Factorial (Common Lisp)

Here's the factorial example; this is the example I tended to use when describing sweet-expressions version 0.1.

Factorial, standard s-expressions (Common Lisp)

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Factorial, Sweet-expressions version 0.1

```
defun factorial (n)
  if (n <= 1)
      1
      n * factorial(n - 1)
```

Factorial, Sweet-expressions version 0.2, infix default

```
defun factorial (n)
  if {n <= 1}
      1
      n * factorial(n - 1)
```

Factorial, Sweet-expressions version 0.2, infix non-default

```
defun factorial (n)
  if {n <= 1}
      1
      {n * factorial{n - 1}}
```

This is nearly a worst-case for infix non-default, which is why initially I resisted it. But in fact, this is a worst case. And it really isn't *that* bad - the f{...} notation makes it tolerable. There isn't even a traditional function call here (something that dominates most examples), which is a tip-off that this example is extreme. Even in this extreme case, there's only one additional time where additional characters must be added to permit infix, and though some thinking is required to notate the passing of "n - 1", it's not too bad.

Factorial (Scheme)

Here's the factorial example, but in Scheme instead. Scheme uses define, not defun, with a slightly different semantic.

Factorial, standard s-expressions (Scheme)

```
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Factorial, Sweet-expressions version 0.1 (Scheme)

```
define factorial(n)
  if (n <= 1)
      1
      n * factorial(n - 1)
```

Factorial, Sweet-expressions version 0.2, infix default (Scheme)

```
define factorial(n)
  if {n <= 1}
      1
      n * factorial(n - 1)
```

Factorial, Sweet-expressions version 0.2, infix non-default (Scheme)

```
define factorial(n)
  if {n <= 1}
      1
      {n * factorial{n - 1}}
```

Same comments as with the Common Lisp version.

Add-if-all-numbers

Here's a Scheme example from Wikipedia - specifically the article "Scheme (programming language)". This example adds an arbitrary list of numbers, and if a non-numeric value is found in the list the procedure is aborted immediately and the constant value #f (false) is returned. This is achieved by capturing the current continuation in the variable exit and using it as an "escape procedure".

Original Scheme

```
(define (add-if-all-numbers lst)
  (call/cc
   (lambda (exit)
     (let loop ((lst lst) (sum 0))
       (if (null? lst) sum
         (if (not (number? (car lst))) (exit #f)
           (+ (car lst) (loop (cdr lst)))))))))
```

Sweet-expressions, version 0.1

```
define add-if-all-numbers(lst)
 call/cc
  lambda (exit)
    let loop ((lst lst) (sum 0))
      if null?(lst)
        sum
        if not(number?(car(lst)))
          exit(#f)
          car(lst) + loop(cdr(lst))
```

add-if-all-numbers, Infix default (rejected)

```
define add-if-all-numbers(lst)
 call/cc
  lambda (exit)
    let loop ((lst lst) (sum 0))
      if null?(lst)
        sum
        if not(number?(car(lst)))
          exit(#f)
          car(lst) + loop(cdr(lst))
```

add-if-all-numbers, Infix non-default

Infix non-default only differs from the previous example in the last line, where we have to explicitly say "use + as infix".

```
define add-if-all-numbers(lst)
 call/cc
  lambda (exit)
    let loop ((lst lst) (sum 0))
      if null?(lst)
        sum
        if not(number?(car(lst)))
          exit(#f)
          {car(lst) + loop(cdr(lst))}
```

Matrix multiply

Matrix multiply example from http://www.scheme.com/tspl2d/examples.html mat-mat-mul multiplies one matrix by another, after verifying that the first matrix has as many columns as the second matrix has rows. I thought a matrix multiply function would show off infix capabilities.

Original Scheme

```
(define mat-mat-mul
   (lambda (m1 m2)
      (let* ((nr1 (matrix-rows m1))
             (nr2 (matrix-rows m2))
             (nc2 (matrix-columns m2))
             (r   (make-matrix nr1 nc2)))
         (if (not (= (matrix-columns m1) nr2))
             (match-error m1 m2))
         (do ((i 0 (+ i 1)))
             ((= i nr1) r)
             (do ((j 0 (+ j 1)))
                 ((= j nc2))
                 (do ((k 0 (+ k 1))
                      (a 0
                         (+ a
                            (* (matrix-ref m1 i k)
                               (matrix-ref m2 k j)))))
                     ((= k nr2)
                      (matrix-set! r i j a))))))))
```

Sweet-expressions, version 0.1

```
define mat-mat-mul
   lambda (m1 m2)
      let* ( (nr1 matrix-rows(m1))
             (nr2 matrix-rows(m2))
             (nc2 matrix-columns(m2))
             (r   make-matrix(nr1 nc2)))
         if not(matrix-columns(m1) = nr2) ; f(infix) handled automatically.
             match-error(m1 m2)
         do ((i 0 (i + 1)))
            ((i = nr1) r)
            do ((j 0 (j + 1)))
               ((j = nc2))
               do ((k 0 (k + 1))
                   (a 0 (a + (matrix-ref(m1 i k) * matrix-ref(m2 k j)))))
                  ((k = nr2) matrix-set!(r i j a))
```

Or, if you use groups:

```
define mat-mat-mul
   lambda (m1 m2)
      let*
        group
          nr1 matrix-rows(m1)
          nr2 matrix-rows(m2)
          nc2 matrix-columns(m2)
          r   make-matrix(nr1 nc2)
        if not(matrix-columns(m1) = nr2) ; f(infix) handled automatically.
          match-error(m1 m2)
        do
          group
            i 0 (i + 1)
          group
            (i = nr1) r
          do
            group
              j 0 (j + 1)
            group
              j = nc2
            do
             group
               k 0 (k + 1)
               a 0 (a + (matrix-ref(m1 i k) * matrix-ref(m2 k j)))
             group
               k = nr2
               matrix-set!(r i j a)
```

Infix default (rejected)

```
define mat-mat-mul
   lambda [m1 m2]
      let* { {nr1 matrix-rows(m1)}
             {nr2 matrix-rows(m2)}
             {nc2 matrix-columns(m2)}
             {r   make-matrix(nr1 nc2)}}
        if not(matrix-columns(m1) = nr2)
            match-error(m1 m2)
        do {{i 0 {i + 1}}}
           {{i = nr1} r}
           do {{j 0 {j + 1}}}
              {{j = nc2}}
              do {{k 0 {k + 1}}
                  {a 0 {a + {matrix-ref(m1 i k) * matrix-ref(m2 k j)}}}}
                 {{k = nr2} matrix-set!(r i j a)}
```

Or, if you use groups:

```
define mat-mat-mul
   lambda [m1 m2]
      let*
        group
          nr1 matrix-rows(m1)
          nr2 matrix-rows(m2)
          nc2 matrix-columns(m2)
          r   make-matrix(nr1 nc2)
        if not(matrix-columns(m1) = nr2) ; f(infix) handled automatically.
          match-error(m1 m2)
        do
          group
            i 0 {i + 1}
          group
            {i = nr1} r
          do
            group
              j 0 {j + 1}
            group
              j = nc2
            do
             group
               k 0 {k + 1}
               a 0 {a + {matrix-ref(m1 i k) * matrix-ref(m2 k j)}}
             group
               k = nr2
               matrix-set!(r i j a)
```

Infix is not Default (Version 0.2)

```
define mat-mat-mul
   lambda [m1 m2]
      let* [ [nr1 matrix-rows(m1)]
             [nr2 matrix-rows(m2)]
             [nc2 matrix-columns(m2)]
             [r   make-matrix(nr1 nc2)]]
        if not{matrix-columns(m1) = nr2} ; f{infix}
            match-error(m1 m2)
        do [[i 0 {i + 1}]]
           [{i = nr1} r]
           do [[j 0 {j + 1}]]
              [{j = nc2}]
              do [[k 0 {k + 1}]
                  [a 0 {a + {matrix-ref(m1 i k) * matrix-ref(m2 k j)}}]]
                 [{k = nr2} matrix-set!(r i j a)]
```

Or, if you use groups:

```
define mat-mat-mul
   lambda [m1 m2]
      let*
        group
          nr1 matrix-rows(m1)
          nr2 matrix-rows(m2)
          nc2 matrix-columns(m2)
          r   make-matrix(nr1 nc2)
        if not{matrix-columns(m1) = nr2} ; f{infix} = f({infix}).
          match-error(m1 m2)
        do
          group
            i 0 {i + 1}
          group
            {i = nr1} r
          do
            group
              j 0 {j + 1}
            group
              {j = nc2}
            do
             group
               k 0 {k + 1}
               a 0 {a + {matrix-ref(m1 i k) * matrix-ref(m2 k j)}}
             group
               {k = nr2}
               matrix-set!(r i j a)
```

Note that in this "infix non-default" example using groups, we have a few cases where we must explicitly state that some operations use infix notation (e.g., {j = nc2} and {k = nr2}), while that was automatically determined by "infix default". On the other hand, in many cases it turns out that it's convenient to group infix operators on a line with {...} anyway, so the automatic detection is used less than you might expect. This is a particularly useful example for discussing the pluses and minuses of making infix default - basically, the automatic rules give fewer benefits than I expected.

Queue

Here's some Common Lisp sample code from lispworks.com (http://www.lispworks.com/documentation/lcl50/ug/ug-22.html).

Original Lisp

Here's the original Common Lisp (slightly reformatted, and with some pieces skipped since the purpose here is just to consider formatting).

```
(in-package "USER")

;; Define a default size for the queue.
(defconstant default-queue-size 100 "Default size of a queue")

;;; The following structure encapsulates a queue. It contains a
;;; simple vector to hold the elements and a pair of pointers to
;;; index into the vector. One is a "put pointer" that indicates
;;; where the next element is stored into the queue. The other is
;;; a "get pointer" that indicates the place from which the next
;;; element is retrieved.
;;; When put-ptr = get-ptr, the queue is empty.
;;; When put-ptr + 1 = get-ptr, the queue is full.
(defstruct (queue (:constructor create-queue)
                  (:print-function queue-print-function))
  (elements #() :type simple-vector) ; simple vector of elements
  (put-ptr 0 :type fixnum) ; next place to put an element
  (get-ptr 0 :type fixnum) ; next place to take an element
  )

(defun queue-next (queue ptr)
  "Increment a queue pointer by 1 and wrap around if needed."
  (let ((length (length (queue-elements queue)))
        (try (the fixnum (1+ ptr))))
       (if (= try length) 0 try)))

(defun queue-get (queue &optional (default nil))
  (check-type queue queue)
  (let ((get (queue-get-ptr queue)) (put (queue-put-ptr queue)))
     (if (= get put) ;; Queue is empty.
        default
        (prog1 (svref (queue-elements queue) get)
               (setf (queue-get-ptr queue) (queue-next queue get))))))

;; Define a function to put an element into the queue. If the
;; queue is already full, QUEUE-PUT returns NIL. If the queue
;; isn't full, QUEUE-PUT stores the element and returns T.
(defun queue-put (queue element)
  "Store ELEMENT in the QUEUE and return T on success or NIL on failure."
  (check-type queue queue)
  (let* ((get (queue-get-ptr queue))
         (put (queue-put-ptr queue))
         (next (queue-next queue put)))
        (unless (= get next) ;; store element
          (setf (svref (queue-elements queue) put) element)
          (setf (queue-put-ptr queue) next) ; update put-ptr
          t))) ; indicate success
```

Infix default (rejected)

```
in-package("USER")

;; Define a default size for the queue.
defconstant(default-queue-size 100 "Default size of a queue")

;;; The following structure encapsulates a queue. It contains a
;;; simple vector to hold the elements and a pair of pointers to
;;; index into the vector. One is a "put pointer" that indicates
;;; where the next element is stored into the queue. The other is
;;; a "get pointer" that indicates the place from which the next
;;; element is retrieved.
;;; When put-ptr = get-ptr, the queue is empty.
;;; When put-ptr + 1 = get-ptr, the queue is full.
defstruct queue(:constructor(create-queue)
                :print-function(queue-print-function))
  elements(#() :type simple-vector) ; simple vector of elements
  put-ptr(0 :type fixnum) ; next place to put an element
  get-ptr(0 :type fixnum) ; next place to take an element

defun queue-next [queue ptr]
  "Increment a queue pointer by 1 and wrap around if needed."
  let
    group
      length length(queue-elements(queue))
      try the(fixnum (1+ ptr))
    if {try = length} 0 try

defun queue-get [queue &optional [default nil]]
  check-type(queue queue)
  let
    group
      get queue-get-ptr(queue)
      put queue-put-ptr(queue)
    if {get = put} ;; Queue is empty.
        default
        prog1
          svref queue-elements(queue) get
          setf queue-get-ptr(queue) queue-next(queue get)

;; Define a function to put an element into the queue. If the
;; queue is already full, QUEUE-PUT returns NIL. If the queue
;; isn't full, QUEUE-PUT stores the element and returns T.
defun queue-put [queue element]
  "Store ELEMENT in the QUEUE and return T on success or NIL on failure."
  check-type(queue queue)
  let*
    group
      get queue-get-ptr(queue)
      put queue-put-ptr(queue)
      next queue-next(queue put)
```

```
    unless {get = next} ;; store element
      setf svref(queue-elements(queue) put) element
      setf queue-put-ptr(queue) next ; update put-ptr
      t ; indicate success
```

Infix is not Default (Version 0.2)

This turns out to be *identical* to the previous case, if square brackets and parentheses are considered equal. However, I've since decided to move to using parentheses rather than square brackets in more places (since in some Lisps square brackets has special meanings). So here it is, using Sweet-Expressions 0.2, using ordinary parentheses:

```
in-package("USER")

;; Define a default size for the queue.
defconstant(default-queue-size 100 "Default size of a queue")

;;; The following structure encapsulates a queue. It contains a
;;; simple vector to hold the elements and a pair of pointers to
;;; index into the vector. One is a "put pointer" that indicates
;;; where the next element is stored into the queue. The other is
;;; a "get pointer" that indicates the place from which the next
;;; element is retrieved.
;;; When put-ptr = get-ptr, the queue is empty.
;;; When put-ptr + 1 = get-ptr, the queue is full.
defstruct queue(:constructor(create-queue)
                :print-function(queue-print-function))
  elements(#() :type simple-vector) ; simple vector of elements
  put-ptr(0 :type fixnum) ; next place to put an element
  get-ptr(0 :type fixnum) ; next place to take an element

defun queue-next (queue ptr)
  "Increment a queue pointer by 1 and wrap around if needed."
  let
    group
      length length(queue-elements(queue))
      try the(fixnum (1+ ptr))
    if {try = length} 0 try

defun queue-get (queue &optional (default nil))
  check-type(queue queue)
  let
    group
      get queue-get-ptr(queue)
      put queue-put-ptr(queue)
    if {get = put} ;; Queue is empty.
        default
        prog1
          svref queue-elements(queue) get
          setf queue-get-ptr(queue) queue-next(queue get)

;; Define a function to put an element into the queue. If the
;; queue is already full, QUEUE-PUT returns NIL. If the queue
;; isn't full, QUEUE-PUT stores the element and returns T.
defun queue-put (queue element)
  "Store ELEMENT in the QUEUE and return T on success or NIL on failure."
  check-type(queue queue)
  let*
    group
      get queue-get-ptr(queue)
      put queue-put-ptr(queue)
      next queue-next(queue put)
```

```
    unless {get = next} ;; store element
      setf svref(queue-elements(queue) put) element
      setf queue-put-ptr(queue) next ; update put-ptr
      t ; indicate success
```

This is an interesting result - here's a whole sequence of code, including some use of infix "=", where the "infix default" and the "infix non-default" case is exactly the same. The reason is that infix is only used in this code as a condition for "if", but it's pretty common to format the condition on the same line as the word "if". As a result, {...} ends up being used in either case.

Macro for Common Lisp

Here's an example from "Lecture Notes: Macros" (http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.html) (on how to do Common Lisp macros). This one shows how to create fully clean Common Lisp macros that don't accidentally capture local variables, using gensym.

Original Lisp

```
(defmacro Square-Sum2 (X Y)
  (let ((First (gensym "FIRST-"))
        (Second (gensym "SECOND-"))
        (Sum (gensym "SUM-")))
    '(let* ((,First ,X)
            (,Second ,Y)
            (,Sum (+ ,First ,Second)))
       (* ,Sum ,Sum))
))
```

Infix default (rejected)

```
defmacro Square-Sum2 (X Y)
  let
    group
      First gensym("FIRST-")
      Second gensym("SECOND-")
      Sum gensym("SUM-")
    'let*
      group
        ,First ,X
        ,Second ,Y
        ,Sum {,First + ,Second}
      ,Sum * ,Sum
```

Infix is not Default (Version 0.2)

```
defmacro Square-Sum2 (X Y)
  let
    group
      First gensym("FIRST-")
      Second gensym("SECOND-")
      Sum gensym("SUM-")
    'let*
      group
        ,First ,X
        ,Second ,Y
        ,Sum {,First + ,Second}
      {,Sum * ,Sum}
```

Again, the last line is different from the infix default case.. but only the last line.

Accumulating factorial

I use the trivial "factorial" example above because, well, it's trivial. But often when writing recursive functions you'll include an accumulator. Here is Dick Gabriel's factorial function (http://www.testing.com/cgi-bin/blog/2004/01/25) that does this, using Common Lisp.

I'll just show the version using "group"; it takes more lines, but I like the look of it.

Original Lisp

```
(defun fact (n)
  (labels ((f (n acc)
             (if (<= n 1) acc (f (- n 1) (* n acc)))))
      (f n 1)))
```

Infix default (rejected)

```
defun fact [n]
  labels
    group
      f [n acc]
        if {n <= 1} acc f({n - 1} {n * acc})
    f n 1
```

Infix is not Default (Version 0.2)

```
defun fact [n]
  labels
    group
      f [n acc]
        if {n <= 1} acc f({n - 1} {n * acc})
    f n 1
```

Again, having infix as a non-default doesn't really matter, the code ends up the same.

Decision Learning Example

Here's a tiny extract of some Decision tree learning code (http://www.cs.cmu.edu/afs/cs/project/theo-11/www/decision-trees.lisp) that accompanies the textbook "Machine Learning," Tom M. Mitchell, McGraw Hill, 1997. "Copyright 1998 Tom M. Mitchell. This code may be freely distributed and

used for any non-commericial purpose, as long as this copyright notice is retained. The author assumes absolutely no responsibility for any harm caused by bugs in the code."

Original Lisp

```
(defun print.tree (tree &optional (depth 0))
  (tab depth)
  (format t "~A~%" (first tree))
  (loop for subtree in (cdr tree) do
        (tab (+ depth 1))
        (format t "= ~A" (first subtree))
        (if (atom (second subtree))
          (format t " => ~A~%" (second subtree))
          (progn (terpri)(print.tree (second subtree) (+ depth 5))))))
(defun tab (n)
  (loop for i from 1 to n do (format t " ")))

(defun classify (instance tree)
  (let (val branch)
    (if (atom tree) (return-from classify tree))
    (setq val (get.value (first tree) instance))
    (setq branch (second (assoc val (cdr tree))))
    (classify instance branch)))

(defun entropy (p)
  (+ (* -1.0 p (log p 2))
     (* -1.0 (- 1 p) (log (- 1 p) 2))))
```

Infix default (rejected)

```
defun print.tree [tree &optional [depth 0]]
  tab depth
  format t "~A~%" first(tree)
  loop for subtree in cdr(tree) do
    tab {depth + 1}
    format t "= ~A" [first subtree]
    if atom(second(subtree))
      format t " => ~A~%" second(subtree)
      progn
        terpri()
        print.tree(second(subtree) {depth + 5})

defun tab [n]
  loop for i from 1 to n do format(t " ")

defun classify [instance tree]
  let
    val branch
    if atom(tree) return-from(classify tree)
    setq val get.value(first(tree) instance)
    setq branch second(assoc(val cdr(tree)))
    classify instance branch

defun entropy [p]
  {-1.0 * p * log(p 2)} + {-1.0 * {1 - p} * log({1 - p} 2)}
```

Infix is not Default (Version 0.2)

```
defun print.tree [tree &optional [depth 0]]
  tab depth
  format t "~A~%" first(tree)
  loop for subtree in cdr(tree) do
    tab {depth + 1}
    format t "= ~A" [first subtree]
    if atom(second(subtree))
      format t " => ~A~%" second(subtree)
      progn
        terpri()
        print.tree(second(subtree) {depth + 5})

defun tab [n]
  loop for i from 1 to n do format(t " ")

defun classify [instance tree]
  let
    val branch
    if atom(tree) return-from(classify tree)
    setq val get.value(first(tree) instance)
    setq branch second(assoc(val cdr(tree)))
    classify instance branch

defun entropy [p]
  {{-1.0 * p * log(p 2)} +
   {-1.0 * {1 - p} * log({1 - p} 2)}}
```

Comments on this example

Here, infix default does produce fewer {...} than infix non-default inside the "entropy" function (which has lots of infix operators). But it is less than you might expect, because as the expression gets long, you tend to want to group things anyway.

We can get fewer {...} in the default infix version by using prefix "+", like this:

```
defun entropy [p]
  +
    -1.0 * p * log(p 2)
    -1.0 * {1 - p} * log({1 - p} 2)
```

Question is, is that enough of an improvement compared to the infix non-default version using this format?:

```
defun entropy [p]
  +
    {-1.0 * p * log(p 2)}
    {-1.0 * {1 - p} * log({1 - p} 2)}
```

Fibonacci Numbers

Original Lisp (Scheme)

Here's an example of the Fibonacci numbers in Scheme by Hanson Char (http://hansonchar.blogspot.com/2006/01/fibonacci-numbers-in-scheme.html), modified. I include a version similar to the original (using cond) and bigger modification of it (using if). I may use the Fibonacci example (using if) as a better example for the ruleset; the factorial example hides ordinary function calls in the infix non-default case, even though they are critical.

```
; Original, using "cond"
(define (fibfast n)
 (cond ((< n 2) n)
       (else (fibup n 2 1 0))))

(define (fibup max count n-1 n-2)
 (cond ((= max count) (+ n-1 n-2))
       (else (fibup max (+ count 1) (+ n-1 n-2) n-1))))


; Using "if"
(define (fibfast n)
 (if (< n 2) n
     (fibup n 2 1 0)))

(define (fibup max count n-1 n-2)
 (if (= max count) (+ n-1 n-2)
     (fibup max (+ count 1) (+ n-1 n-2) n-1)))
```

Infix default (rejected)

```
; Original, using "cond"
define fibfast(n)
  cond
    {n < 2} n
    else       fibup(n 2 1 0)

define fibup(max count n-1 n-2)
  cond
    {max = count} {n-1 + n-2}
    else          fibup(max {count + 1} {n-1 + n-2} n-1)


; Using "if"
define fibfast(n)
  if {n < 2}
    n
    fibup(n 2 1 0)

define fibup(max count n-1 n-2)
  if {max = count}
    n-1 + n-2
    fibup(max {count + 1} {n-1 + n-2} n-1)
```

Infix is not Default (Version 0.2)

```
; Original, using "cond"
define fibfast(n)
  cond
    {n < 2} n
    else       fibup(n 2 1 0)

define fibup(max count n-1 n-2)
  cond
    {max = count} {n-1 + n-2}
    else          fibup(max {count + 1} {n-1 + n-2} n-1)


; Using "if"
define fibfast(n)
  if {n < 2}
    n
    fibup(n 2 1 0)

define fibup(max count n-1 n-2)
  if {max = count}
    {n-1 + n-2}
    fibup(max {count + 1} {n-1 + n-2} n-1)
```

Again, the infix and non-infix are nearly identical; only one line (the second from the last) is different (to make "+" work as infix).

ACL2

ACL2 is an interactive mechanical theorem prover (http://www.cs.utexas.edu/users/moore/acl2/) designed for use in modeling hardware and software and proving properties about those models. It is released under the GPL license.

ACL2 is powerful, but it's often avoided specifically because many people find its Lisp notation to be user-hostile. For example, David Duffy's book "Principles of Automated Theorem Proving" (1991) devotes a whole chapter to the Boyer-Moore Theorem Prover (ACL2 is the latest version of this series). The chapter specifically states that one of ACL2's key problems is the "difficulty of reading the LISP-like

prefix notation" and that "To improve readability here, this notation will often be abused to include the use of prefix and infix symbols" (page 176-177). For the rest of the chapter, the author modifies ACL2 input and output, instead of showing actual input, so that readers could understand what is going on. In contrast, in a different chapter the author did not modify Prolog's notation, because Prolog's notation is much easier to read for those trained in traditional mathematics or other programming languages.

Original Lisp

This is a simple example from their "flying demo" (http://www.cs.utexas.edu/users/moore/publications/flying-demo/script.html#next6).

```
(thm (implies (and (not (endp x))
                           (endp (cdr x))
                           (integerp n)
                           (<= 0 n)
                           (rationalp u))
                      (< (* (len x) u) (+ u n 3))))
```

Infix default (rejected)

There are different ways to format this. One way emphasizes indentation:

```
  thm
    implies
      and not(endp(x))
           endp(cdr(x))
           integerp(n)
           0 <= n
           rationalp(u)
      {len(x) * u} < {u + n + 3}
```

You can exploit the fact that infix is by default, and use the /.../ special rule for infix operators that are not punctuation. Here's one way:

```
thm {
    {not(endp(x)) and endp(cdr(x)) and
     integerp(n)  and {0 <= n} and rationalp(u)}
  /implies/
    { {len(x) * u} < {u + n + 3} } }
```

Here's another way (note that here, we don't have to disable indentation just to use "implies" as an infix operator):

```
thm
  not(endp(x)) and endp(cdr(x)) and integerp(n) and {0 <= n} and rationalp(u)
  /implies/
  {len(x) * u} < {u + n + 3}
```

Infix is not Default (Version 0.2)

There are different ways to format this. One way emphasizes indentation:

```
thm
  implies
    and not(endp(x))
         endp(cdr(x))
         integerp(n)
         {0 <= n}
         rationalp(u)
    { {len(x) * u} < {u + n + 3} }
```

Another way emphasizes infix:

```
thm {
  {not(endp(x)) and endp(cdr(x)) and integerp(n) and {0 <= n}
    and rationalp(u)}
  implies
    { {len(x) * u} < {u + n + 3} } }
```

The indented version shows a little more difference with the infix default vs. non-default. In addition, here there *is* a way to use "infix default" that the infix non-default can't really do (infix default can have an infix operator marked with /.../ without disabling indentation). On the other hand, it's odd enough that it's not an expecially convincing argument.

This is a case where if the "immediate completion" rules were in force, you would need to beware of a single-term starting on the left-hand-side.

Longer ACL2 Example

```
(defun rev3 (x)
  (cond
   ((endp x)
    nil)
   ((endp (cdr x))
    (list (car x)))
   (t
    (let* ((b@c (cdr x))
           (c@rev-b (rev3 b@c)) ; note recursive call of rev3
           (rev-b (cdr c@rev-b))
           (b (rev rev-b))      ; note call of rev
           (a (car x))
           (a@b (cons a b))
           (rev-b@a (rev a@b))  ; note call of rev
           (c (car c@rev-b))
           (c@rev-b@a (cons c rev-b@a)))
      c@rev-b@a))))
```

So, let's see if we can pretty this up with sweet-expressions, using the "infix is not the default" rules.

Infix is not Default (Version 0.2)

```
defun rev3 (x)
  cond
   endp(x)      nil
   endp(cdr(x)) list(car(x))
   t
     let*
       group
         b@c         cdr(x)
         c@rev-b     rev3(b@c)    ; note recursive call of rev3
         rev-b       cdr(c@rev-b)
         b           rev(rev-b)   ; note call of rev
         a           car(x)
         a@b         cons(a b)
         rev-b@a     rev(a@b)     ; note call of rev
         c           car(c@rev-b)
         c@rev-b@a   cons(c rev-b@a)
       c@rev-b@a
```

BitC Example

BitC (http://www.bitc-lang.org/) is, according to its creators, "a systems programming language that combines the ``low level'' nature of C with the semantic rigor of Scheme or ML. BitC was designed by careful selection and exclusion of language features in order to support proving properties (up to and including total correctness) of critical systems programs." See the BitC specification for more information (http://www.bitc-lang.org/docs/bitc/spec.html). BitC is in development, the following example is from the version 0.10+ (June 17, 2006) specification. The BitC reader is actually not quite a standard s-expression reader; in particular, it treats ":" specially. Hopefully, a sweet-expression reader will be even better.

Original BitC

Here's the original BitC. I've cheated slightly with the definition of ">", pulling it out of context. One complication: BitC's reader is not a "pure" s-expression reader; it handles ":" specially.

```
(deftypeclass
  (forall ((Eql 'a)) (Ord 'a))
  < : (fn ('a 'a) 'a))

(define (> x y)
  (not (or (< x y) (== x y))))

(define (fact x:int32)
  (cond ((< x 0) (- (fact (- x))))
        ((= x 0) 1)
        (otherwise
          (* x (fact (- x 1))))))
```

Infix default (rejected)

```
deftypeclass
  forall (Eql('a)) Ord('a)
  < : fn(('a 'a) 'a)

define >(x y)
  not({x < y} or {x == y})
```

I've given up on infix default, so I haven't tried to handle "fact". This is just as well; ":" appears to sometimes require infix processing, and sometimes it doesn't.

Infix is not Default (Version 0.2)

```
deftypeclass
  forall (Eql('a)) Ord('a)
  < : fn(('a 'a) 'a)

define >(x y)
  not{ {x < y} or {x == y} }

define fact{x : int32}
  cond
   {x < 0}     -(fact(-(x)))
   {x = 0}     1
   otherwise   {x * fact{x - 1}}
```

The infix and non-infix default are slightly different if you follow typical (though that isn't necessary; the non-infix default one would also work). Basically, "not" can be an ordinary function call, since the "or" inside the "not" is automatically detected as being an infix operator. It could be argued that the infix version is slightly nicer to read here. However, since f{...} is an accepted abbreviation for f({...}), the non-infix default version is not all *that* bad.

Since "deftypeclass" is on a line by itself, if the "immediate completion" rules were in force it needs to *not* begin the line (press at least one space). Otherwise, it would be executed immediately. But the "immediate completion" rules have been removed, so this is not an issue.

The ":" in many BitC contexts as a "type assertion" operator. The BitC reader handles ":" very specially. Instead of handling ":" specially, we can treat ":" as just another infix operator taking two parameters: the object and its type. Which means that instead of a special-case reader, we can use a general-case reader.

PVS

PVS is a verification system (http://pvs.csl.sri.com/), i.e., "a specification language integrated with support tools and a theorem prover". It is released under the GNU GPL license. It's implemented using Common Lisp. PVS implements its own specification language that supports infix, etc., and that doesn't need replacing. However, when proving theories, you must interact with a essentially a Lisp read-eval-print loop, and that *does* use ordinary s-expression syntax. This would be especially valuable when defining new strategies.

Original Lisp

Here's a definition for non-default strategy called "stew". I can't remember where I got this; I think I got this definition from elsewhere and then tweaked it.

Oh, one warning: both "if" and "then" are *commands* in this notation; an "if" does *not* have a "then" keyword. I mention this, because in this example it can look confusing.

```
 (defstep stew (&optional lazy-match (if-match t) (defs !) rewrites theories
                exclude (updates? t) &rest lemmas)
   (then
     (if lemmas
       (let ((lemmata (if (listp lemmas) lemmas (list lemmas)))
             (x `(then ,@(loop for lemma in lemmata append `((skosimp*)(use ,le
mma))))))
           x)
        (skip))
      (if lazy-match
        (then (grind$ :if-match nil :defs defs :rewrites rewrites
                      :theories theories :exclude exclude :updates? updates?)
              (reduce$ :if-match if-match :updates? updates?))
        (grind$ :if-match if-match :defs defs :rewrites rewrites
                      :theories theories :exclude exclude :updates? updates?))
)
 "Does a combination of (lemma) and (grind)."
 "~%Grinding away with the supplied lemmas,")
```

Infix default (rejected)

```
defstep stew (&optional lazy-match (if-match t) (defs !) rewrites theories
              exclude (updates? t) &rest lemmas)
  then
    if lemmas
      let
        group
          lemmata
            if listp(lemmas) lemmas list(lemmas)
          x `[then ,@[loop for lemma in lemmata append
                       `[skosimp*() use(,lemma)]]]
        x
      skip()
    if lazy-match
       then
         grind$(:if-match nil :defs defs :rewrites rewrites
                 :theories theories :exclude exclude :updates? updates?)
         reduce$ :if-match if-match :updates? updates?
       grind$(:if-match if-match :defs defs :rewrites rewrites
              :theories theories :exclude exclude :updates? updates?)
  "Does a combination of (lemma) and (grind)."
  "~%Grinding away with the supplied lemmas,"
```

Infix is not Default (Version 0.2)

```
defstep stew (&optional lazy-match (if-match t) (defs !) rewrites theories
              exclude (updates? t) &rest lemmas)
  then
    if lemmas
      let
        group
          lemmata
            if listp(lemmas) lemmas list(lemmas)
          x `[then ,@[loop for lemma in lemmata append
                       `[skosimp*() use(,lemma)]]]
        x
      skip()
    if lazy-match
       then
         grind$(:if-match nil :defs defs :rewrites rewrites
                 :theories theories :exclude exclude :updates? updates?)
         reduce$ :if-match if-match :updates? updates?
       grind$(:if-match if-match :defs defs :rewrites rewrites
              :theories theories :exclude exclude :updates? updates?)
  "Does a combination of (lemma) and (grind)."
  "~%Grinding away with the supplied lemmas,"
```

No difference between the infix default and non-default.

Emacs Lisp

Here's an example of emacs Lisp, this time from a page on emacs Lisp by Xah Lee (http://xahlee.org/emacs/elisp_examples.html). (Note: Emacs Lisp's variable scoping is dynamic, a fossil from very old versions of Lisp. Scheme and Common Lisp's variable scope is not.)

Original Lisp

```
(defun replace-html-chars (start end)
  "Replace '<' to '&lt;' and other chars in HTML.
This works on the current selection."
  (interactive "r")
  (save-restriction
    (narrow-to-region start end)
    (goto-char (point-min))
    (while (search-forward "&" nil t) (replace-match "&amp;" nil t))
    (goto-char (point-min))
    (while (search-forward "<" nil t) (replace-match "&lt;" nil t))
    (goto-char (point-min))
    (while (search-forward ">" nil t) (replace-match "&gt;" nil t))
    )
  )
```

Infix default (rejected)

```
defun replace-html-chars (start end)
  "Replace '<' to '&lt;' and other chars in HTML.
This works on the current selection."
  interactive("r")
  save-restriction
    narrow-to-region start end
    goto-char point-min()
    while search-forward("&" nil t) replace-match("&amp;" nil t)
    goto-char point-min()
    while search-forward("<" nil t) replace-match("&lt;" nil t)
    goto-char point-min()
    while search-forward(">" nil t) replace-match("&gt;" nil t)
```

Infix is not Default (Version 0.2)

```
defun replace-html-chars (start end)
  "Replace '<' to '&lt;' and other chars in HTML.
This works on the current selection."
  interactive("r")
  save-restriction
    narrow-to-region start end
    goto-char point-min()
    while search-forward("&" nil t) replace-match("&amp;" nil t)
    goto-char point-min()
    while search-forward("<" nil t) replace-match("&lt;" nil t)
    goto-char point-min()
    while search-forward(">" nil t) replace-match("&gt;" nil t)
```

The infix default and non-infix default versions are identical in this case.

Find (AutoCAD Lisp / AutoLisp)

AutoCAD includes its own Lisp languages, Autolisp. From the Free Autolisp routines (http://members.aol.com/autolisper/software.htm) I arbitrarily chose the "Find" program, whose purpose is to "Find text in [a] drawing field".

Original Lisp

Here's the original code, as provided:

```
(DEFUN C:FIND ( )
    (SETQ SA(GETSTRING T "\nEnter string for search parameter: "))
    (SETQ AR(SSGET "X" (LIST(CONS 1 SA))))
    (IF(= AR NIL)(ALERT "This string does not exist"))
    (SETQ SB(SSLENGTH AR))

    (C:CONT)
    )
(DEFUN C:CONT ()
   (SETQ SB(- SB 1))

   (SETQ SC(SSNAME AR SB))
   (SETQ SE(ENTGET SC))
   (SETQ SJ(CDR(ASSOC 1 SE)))
   (IF(= SJ SA)(PROGN
      (SETQ H(CDR(ASSOC 10 SE)))
      (SETQ X1(LIST(- (CAR H) 50)(- (CADR H)50)))
      (SETQ X2(LIST(+ 50(CAR H))(+ 50 (CADR H))))
      (COMMAND "ZOOM" "W" X1 X2 ))(C:CONT)
      )
 (IF(= SB 0)(ALERT "END OF SELECTIONS"))
(SETQ A(+ SB 1))
(SETQ A(RTOS A 2 0))
(SETQ A(STRCAT "\nThere are  selections Enter CONT to advance to next"))
(IF(= SB 0)(EXIT))
(PRINC A)
(PRINC)
   )
```

That's so hideously formatted that we can create a much more readable version without needing a new reader. We'll stick to uppercase, so that we can see that the improvement is unrelated to using uppercase or lowercase:

```
(DEFUN C:FIND ()
  (SETQ SA (GETSTRING T "\nEnter string for search parameter: "))
  (SETQ AR (SSGET "X" (LIST (CONS 1 SA))))
  (IF (= AR NIL) (ALERT "This string does not exist"))
  (SETQ SB (SSLENGTH AR))
  (C:CONT))

(DEFUN C:CONT ()
  (SETQ SB (- SB 1))
  (SETQ SC (SSNAME AR SB))
  (SETQ SE (ENTGET SC))
  (SETQ SJ (CDR (ASSOC 1 SE)))
  (IF (= SJ SA)
    (PROGN
      (SETQ H (CDR (ASSOC 10 SE)))
      (SETQ X1 (LIST (- (CAR H) 50) (- (CADR H) 50)))
      (SETQ X2 (LIST (+ 50 (CAR H)) (+ 50 (CADR H))))
      (COMMAND "ZOOM" "W" X1 X2))
    (C:CONT))
  (IF (= SB 0) (ALERT "END OF SELECTIONS"))
  (SETQ A (+ SB 1))
  (SETQ A (RTOS A 2 0))
  (SETQ A
    (STRCAT "\nThere are  selections Enter CONT to advance to next"))
  (IF (= SB 0) (EXIT))
  (PRINC A)
  (PRINC))
```

Infix is not Default (Version 0.2)

Okay, now we'll use sweet-expressions 0.2, with infix non-default. Again, we'll keep it in all uppercase, so that you won't be misled by a difference in case. Notice that even with all-upper-case, it still is easier to follow than the original:

```
DEFUN C:FIND ()
  SETQ SA GETSTRING(T "\nEnter string for search parameter: ")
  SETQ AR SSGET("X" LIST(CONS(1 SA)))
  IF {AR = NIL} ALERT("This string does not exist")
  SETQ SB SSLENGTH(AR)
  C:CONT()

DEFUN C:CONT ()
  SETQ SB {SB - 1}
  SETQ SC SSNAME(AR SB)
  SETQ SE ENTGET(SC)
  SETQ SJ CDR(ASSOC(1 SE))
  IF {SJ = SA}
    PROGN
      SETQ H CDR(ASSOC(10 SE))
      SETQ X1 LIST({CAR(H) - 50} {CADR(H) - 50})
      SETQ X2 LIST({50 + CAR(H)} {50 + CADR(H)})
      COMMAND("ZOOM" "W" X1 X2)
    C:CONT()
  IF {SB = 0} ALERT("END OF SELECTIONS")
  SETQ A {SB + 1}
  SETQ A RTOS(A 2 0)
  SETQ A
    STRCAT "\nThere are  selections Enter CONT to advance to next"
  IF {SB = 0} EXIT()
  PRINC A
  PRINC()
```

Today most people use lowercase, so let's see how this looks with that one change:

```
defun c:find ()
  setq sa getstring(t "\nEnter string for search parameter: ")
  setq ar ssget("x" list(cons(1 sa)))
  if {ar = nil} alert("This string does not exist")
  setq sb sslength(ar)
  c:cont()

defun c:cont ()
  setq sb {sb - 1}
  setq sc ssname(ar sb)
  setq se entget(sc)
  setq sj cdr(assoc(1 se))
  if {sj = sa}
    progn
      setq h cdr(assoc(10 se))
      setq x1 list({car(h) - 50} {cadr(h) - 50})
      setq x2 list({50 + car(h)} {50 + cadr(h)})
      command("ZOOM" "W" x1 x2)
    c:cont()
  if {sb = 0} alert("END OF SELECTIONS")
  setq a {sb + 1}
  setq a rtos(a 2 0)
  setq a
    strcat "\nThere are  selections Enter CONT to advance to next"
  if {sb = 0} exit()
  princ a
  princ()
```

Kalotan puzzle (Scheme)

"Teach Yourself Scheme in Fixnum days" (http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html) includes a way to solve the Kalotan puzzle solution using Scheme (http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-16.html#node_sec_14.4.1). It depends on the "amb" function, described further in the book. It has a large number of "ands" and "xors" which can be expressed using infix, which is interesting and useful for trying out variations.

Original Lisp

Here's the original Lisp:

```
(define solve-kalotan-puzzle
  (lambda ()
    (let ((parent1 (amb 'm 'f))
          (parent2 (amb 'm 'f))
          (kibi (amb 'm 'f))
          (kibi-self-desc (amb 'm 'f))
          (kibi-lied? (amb #t #f)))
      (assert
       (distinct? (list parent1 parent2)))
      (assert
       (if (eqv? kibi 'm)
           (not kibi-lied?)))
      (assert
       (if kibi-lied?
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'f))
            (and (eqv? kibi-self-desc 'f)
                 (eqv? kibi 'm)))))
      (assert
       (if (not kibi-lied?)
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'm))
            (and (eqv? kibi-self-desc 'f)
                 (eqv? kibi 'f)))))
      (assert
       (if (eqv? parent1 'm)
           (and
            (eqv? kibi-self-desc 'm)
            (xor
             (and (eqv? kibi 'f)
                  (eqv? kibi-lied? #f))
             (and (eqv? kibi 'm)
                  (eqv? kibi-lied? #t))))))
      (assert
       (if (eqv? parent1 'f)
           (and
            (eqv? kibi 'f)
            (eqv? kibi-lied? #t))))
      (list parent1 parent2 kibi))))

(solve-kalotan-puzzle)
```

Infix non-default

Here's one way to convert this to infix non-default, emphasizing the use of indentation and function calls using prefixed():

```
define solve-kalotan-puzzle
  lambda []
    let [[parent1 amb('m 'f)]
         [parent2 amb('m 'f)]
         [kibi amb('m 'f)]
         [kibi-self-desc amb('m 'f)]
         [kibi-lied? amb(#t #f)]]
      assert
       distinct?(list(parent1 parent2))
      assert
       if eqv?(kibi 'm)
           not(kibi-lied?)
      assert
       if kibi-lied?
          xor
            and eqv?(kibi-self-desc 'm)
                eqv?(kibi 'f)
            and eqv?(kibi-self-desc 'f)
                eqv?(kibi 'm)
      assert
       if not(kibi-lied?)
          xor
            and eqv?(kibi-self-desc 'm)
                eqv?(kibi 'm)
            and eqv?(kibi-self-desc 'f)
                eqv?(kibi 'f)
      assert
       if eqv?(parent1 'm)
          and
            eqv?(kibi-self-desc 'm)
            xor
             and eqv?(kibi 'f)
                 eqv?(kibi-lied? #f)
             and eqv?(kibi 'm)
                 eqv?(kibi-lied? #t)
      assert
       if eqv?(parent1 'f)
          and
            eqv?(kibi 'f)
            eqv?(kibi-lied? #t)
      list(parent1 parent2 kibi)

solve-kalotan-puzzle()
```

Let's use more infix operations; note that there's no requirement that we use infix everywhere:

```
define solve-kalotan-puzzle
  lambda []
    let [[parent1 amb('m 'f)]
         [parent2 amb('m 'f)]
         [kibi amb('m 'f)]
         [kibi-self-desc amb('m 'f)]
         [kibi-lied? amb(#t #f)]]
      assert
       distinct?(list(parent1 parent2))
      assert
       if eqv?(kibi 'm)
           not(kibi-lied?)
      assert
       if kibi-lied?
          xor
            {eqv?(kibi-self-desc 'm) and eqv?(kibi 'f)}
            {eqv?(kibi-self-desc 'f) and eqv?(kibi 'm)}
      assert
       if not(kibi-lied?)
          xor
            {eqv?(kibi-self-desc 'm) and eqv?(kibi 'm)}
            {eqv?(kibi-self-desc 'f) and eqv?(kibi 'f)}
      assert
       if eqv?(parent1 'm)
          and
            eqv?(kibi-self-desc 'm)
            xor
             {eqv?(kibi 'f) and eqv?(kibi-lied? #f)}
             {eqv?(kibi 'm) and eqv?(kibi-lied? #t)}
      assert
       if eqv?(parent1 'f)
          {eqv?(kibi 'f) and eqv?(kibi-lied? #t)}
      list(parent1 parent2 kibi)

solve-kalotan-puzzle()
```

Now let's add use the "group" command to get rid of some more brackets:

```
define solve-kalotan-puzzle
  lambda []
    let
      group
        parent1         amb('m 'f)
        parent2         amb('m 'f)
        kibi            amb('m 'f)
        kibi-self-desc  amb('m 'f)
        kibi-lied?      amb(#t #f)
      assert
       distinct?(list(parent1 parent2))
      assert
       if eqv?(kibi 'm)
           not(kibi-lied?)
      assert
       if kibi-lied?
          xor
            {eqv?(kibi-self-desc 'm) and eqv?(kibi 'f)}
            {eqv?(kibi-self-desc 'f) and eqv?(kibi 'm)}
      assert
       if not(kibi-lied?)
          xor
            {eqv?(kibi-self-desc 'm) and eqv?(kibi 'm)}
            {eqv?(kibi-self-desc 'f) and eqv?(kibi 'f)}
      assert
       if eqv?(parent1 'm)
          and
            eqv?(kibi-self-desc 'm)
            xor
             {eqv?(kibi 'f) and eqv?(kibi-lied? #f)}
             {eqv?(kibi 'm) and eqv?(kibi-lied? #t)}
      assert
       if eqv?(parent1 'f)
          {eqv?(kibi 'f) and eqv?(kibi-lied? #t)}
      list(parent1 parent2 kibi)

solve-kalotan-puzzle()
```

And as a test, let's use even more of the infix operators. Frankly, I think the previous version is easier to follow, though this version is more similar to how it'd be done in many other languages:

```
define solve-kalotan-puzzle
  lambda []
    let
      group
        parent1         amb('m 'f)
        parent2         amb('m 'f)
        kibi            amb('m 'f)
        kibi-self-desc  amb('m 'f)
        kibi-lied?      amb(#t #f)
      assert
       distinct?(list(parent1 parent2))
      assert
       if eqv?(kibi 'm)
           not(kibi-lied?)
      assert
       if kibi-lied?
          {{eqv?(kibi-self-desc 'm) and eqv?(kibi 'f)}   xor
           {eqv?(kibi-self-desc 'f) and eqv?(kibi 'm)}}
      assert
       if not(kibi-lied?)
          {{eqv?(kibi-self-desc 'm) and eqv?(kibi 'm)}   xor
           {eqv?(kibi-self-desc 'f) and eqv?(kibi 'f)}}
      assert
       if eqv?(parent1 'm)
          {eqv?(kibi-self-desc 'm) and
            {{eqv?(kibi 'f) and eqv?(kibi-lied? #f)}     xor
             {eqv?(kibi 'm) and eqv?(kibi-lied? #t)}}}
      assert
       if eqv?(parent1 'f)
          {eqv?(kibi 'f) and eqv?(kibi-lied? #t)}
      list(parent1 parent2 kibi)

solve-kalotan-puzzle()
```

"Readable" page

On the "readable" front page I show a table of examples, comparing S-expressions and sweet-expressions. All of these examples use Scheme.

Original Table (Sweet-expressions 0.1)

Sweet-expression 0.1 (Ugly) S-expression

```
define factorial(n)
   if (n <= 1)
       1
       n * factorial(n - 1)
substring("Hello" (1 + 1)
          string-length("Hello"))
define move-n-turn(angle)
   tortoise-move(100)
   tortoise-turn(angle)
if (0 <= 5 <= 10)
   display("True\n")
   display("Uh oh\n")
define int-products(x y)
  if (x = y)
    x
    x * int-products((x + 1) y)
int-products(3 5)
3 + 4
(2 + 3 + (4 * 5) + 7.1)
*(2 3 4 5)
```

```
(define (factorial n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
(substring "Hello" (+ 1 1)
           (string-length "Hello"))
(define (move-n-turn angle)
   (tortoise-move 100)
   (tortoise-turn angle))
(if (<= 0 5 10)
   (display "True\n")
   (display "Uh oh\n"))
(define (int-products x y)
  (if (= x y)
      x
      (* x (int-products (+ x 1) y))))
(int-products 3 5)
(+ 3 4)
(+ 2 3 (* 4 5) 7.1)
(* 2 3 4 5)
```

New Table (Sweet-expressions 0.2, infix non-default)

Sweet-expression 0.2 (Ugly) S-expression

```
define factorial(n)
   if {n <= 1}
       1
       {n * factorial{n - 1}}
substring("Hello" {1 + 1}
          string-length("Hello"))
define move-n-turn(angle)
   tortoise-move(100)
   tortoise-turn(angle)
if {0 <= 5 <= 10}
   display("True\n")
   display("Uh oh\n")
define int-products(x y)
  if {x = y}
    x
    {x * int-products( {x + 1} y )}
int-products(3 5)
{3 + 4}
{2 + 3 + {4 * 5} + 7.1}
*(2 3 4 5)  or  {2 * 3 * 4 * 5}
```

```
(define (factorial n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
(substring "Hello" (+ 1 1)
           (string-length "Hello"))
(define (move-n-turn angle)
   (tortoise-move 100)
   (tortoise-turn angle))
(if (<= 0 5 10)
   (display "True\n")
   (display "Uh oh\n"))
(define (int-products x y)
  (if (= x y)
      x
      (* x (int-products (+ x 1) y))))
(int-products 3 5)
(+ 3 4)
(+ 2 3 (* 4 5) 7.1)
(* 2 3 4 5)
```

See the readable Lisp page for more. (http://www.dwheeler.com/readable) The older paper Readable s-expressions and sweet-expressions (https://www.dwheeler.com/readable/readable-s-expressions.html) describes the

rationale and background for sweet-expressions version 0.1, and gives useful information on related approaches.

SUO-KIF

Standard Upper Ontology Knowledge Interchange Format (SUO-KIF) (http://sigmakee.cvs.sourceforge.net/*checkout*/sigmakee/sigma/suo-kif.pdf) is a language designed for use in the authoring and interchange of knowledge, e.g., by the Suggested Upper Merged Ontology (SUMO) (http://www.ontologyportal.org/).

Original Lisp

Here's the original Lisp of a trivial example, "All farmers like tractors":

```
(forall (?F ?T)
  (=>
    (and
       (instance ?F Farmer)
       (instance ?T Tractor))
    (likes ?F ?T)))
```

Infix non-default

Without using any infix operators, we can get a nicer result:

```
forall (?F ?T)
  =>
    and
       instance(?F Farmer)
       instance(?T Tractor)
    likes ?F ?T
```

However, both "and" and "=>" (implies) are traditionally used as infix operators, rather than prefix operators. Using {...}, we can use them in that traditional manner:

```
forall (?F ?T)
  {{instance(?F Farmer) and instance(?T Tractor)} => likes(?F ?T)}
```

or

```
forall (?F ?T)
  { {instance(?F Farmer) and instance(?T Tractor)}
    =>
    likes(?F ?T)}
```

Scsh (Scheme Shell)

Scheme shell (http://www.scsh.net/about/what.html) is an implementation of Scheme, plus a high-level process notation for doing shell-script-like tasks (running programs, establishing pipelines, and I/O redirection).

Scsh's documentation (section 1.4) (http://www.scsh.net/docu/docu.html) notes that scsh is (currently) "primarily designed for the writing of shell scripts - programming. It is not a very comfortable system for interactive command use: the current release lacks job control, command-line editing, a terse, convenient command syntax, and it does not read in an initialisation file analogous to .login or .profile." Most of these limitations can be easily fixed by reusing existing code; e.g., job control code can be copied from other shells, there are trivial libraries (such as readline) that implement command-line editing, and reading an initialization file is trivially done. But the "convenient command syntax" is a serious stumbling block to using scsh as a shell, and that is *not* trivially fixed by simply reusing arbitrary code. It *can* be fixed by using sweet-expressions.

Original Lisp

In traditional shells, you might write a command like this:

```
gunzip < paper.tex.gz | detex | spell | lpr -Ppulp &
```

In scsh, this could be written as:

```
(& (| (gunzip) (detex) (spell) (lpr -Ppulp)) ; background a pipeline
    (< paper.tex.gz))                         ; with this redirection
```

Infix non-default

In sweet-expressions, the same expression of scsh can be written as:

```
&
  | gunzip() detex() spell() lpr(-Ppulp) ; background a pipeline
  < paper.tex.gz                         ; with this redirection
```

or as:

```
&
  {gunzip() | detex() | spell() | lpr(-Ppulp)} ; background a pipeline
  < paper.tex.gz                               ; with this redirection
```

I think it's better if punctuation-only names are used primarily for infix operators; it helps with consistency. At the least, I'd rename "&" into the synonym "background". Then it becomes:

```
background
  {gunzip() | detex() | spell() | lpr(-Ppulp)} ; background a pipeline
  < paper.tex.gz                               ; with this redirection
```

Arc

Paul Graham is developing a new dialect of Lisp named Arc; Paul Graham's Arc site (http://www.paulgraham.com/arc.html) and Arclanguage.org (http://arclanguage.org/) have more information. Semantically, it's a lot like a special combination of Common Lisp and Scheme. For example, like Common Lisp, both "false" and "empty list" are nil (Scheme has separate objects for false and the empty list). But like Scheme, Arc has a single namespace (it's a Lisp-1), as compared with Common Lisp (which is a Lisp-2). Arc has some cool ideas, but Paul Graham seems to be very busy on other things, so an informal community called 'Anarki' (http://arcfn.com/2008/02/git-and-anarki-arc-repository-brief.html) has formed to evolve Arc. (It uses "=" for assignment, which I think is unfortunate because "=" is often confused with is-equal-to, but anyway...). A good place to start is the Arc tutorial (http://ycombinator.com/arc/tut.txt).

In Arc, [...] has a special meaning. More specifically, "[... _ ...]" is an abbreviation for "(fn (_) (... _ ...))". For example:

```
arc> (map [+ _ 10] '(1 2 3))
(11 12 13)
```

This makes me think that sweet-expressions should *not* mandate that [...] be equivalent to (...), necessarily, in sweet-expressions. Instead, the meaning of [...] may be somewhat language-dependent in sweet-expressions; a Scheme using sweet-expressions might consider [...] equivalent to (...), since the normal Scheme R6 does, but it doesn't necessarily follow that this must be true for all languages. If you disable sweet-expression's definition that [...] is the same as (...), it looks like Arc and sweet-expressions could be easily used together.

Original Lisp

The Arc tutorial (http://ycombinator.com/arc/tut.txt) has several really tiny examples; let's pick out a few and put them in one place:

```
(+ (+ 1 2) (+ 3 (+ 4 5)))
(def average (x y)
       (/ (+ x y) 2))
(let x 10
  (while (> x 5)
    (= x (- x 1))
    (pr x)))
(mac repeat (n . body)
  `(for ,(uniq) 1 ,n ,@body))
(def firstn (n xs)
  (if (and (> n 0) xs)
      (cons (car xs) (firstn (- n 1) (cdr xs)))
      nil))
(def nthcdr (n xs)
  (if (> n 0)
      (nthcdr (- n 1) (cdr xs))
      xs))
(def tuples (xs (o n 2))
  (if (no xs)
      nil
      (cons (firstn n xs)
            (tuples (nthcdr n xs) n))))

(def mylen (xs)
      (if (no xs)
          0
          (+ 1 (mylen (cdr xs)))))

(mac n-of (n expr)
  (w/uniq ga
    `(let ,ga nil
       (repeat ,n (push ,expr ,ga))
       (rev ,ga))))
(if
  condition1 result1
  condition2 result2
  default-result)
```

Infix non-default

Here are the sweet-expression equivalents:

```
{{1 + 2} + {3 + {4 + 5}}}
def average (x y)
       {{x + y} / 2}
let x 10
  while {x > 5}
    {x = {x - 1}}
    pr x
mac repeat (n . body)
  `for ,uniq() 1 ,n ,@body
def firstn (n xs)
  if {{n > 0} and xs}
      cons car(xs) firstn({n - 1} cdr(xs))
      nil
def nthcdr (n xs)
  if {n > 0}
      nthcdr {n - 1} cdr(xs)
      xs
def tuples (xs (o n 2))
  if no(xs)
      nil
      cons firstn(n xs)
           tuples nthcdr(n xs) n

def mylen (xs)
      if no(xs)
          0
          {1 + mylen(cdr(xs))}

mac n-of (n expr)
  w/uniq ga
    `let ,ga nil
       repeat ,n push(,expr ,ga)
       rev ,ga
if(
  condition1 result1
  condition2 result2
  default-result)
```

Note the macro "n-of"; even though it's short, the original required 4 closing parentheses to complete it, while the sweet-expression required none.

Indentation processing doesn't work as easily when there's an implied pairing of list elements that isn't actually in the list structure, like Arc's "if". See the idea, below, for more about this.

RTL (GCC Register Transfer Language)

The GNU Compiler Collection (GCC) has an internal structure for representing programs (at a low level), the GCC Register Transfer Language (RTL) (http://gcc.gnu.org/onlinedocs/gccint/RTL.html). It's often useful to print out RTL; it's useful for debugging, and other tools use it too (such as RTL-check (http://rtlcheck.sourceforge.net/) and Egypt (http://www.gson.org/egypt/egypt.html). Here's a summary of some RTL semantics. (http://www.cs.berkeley.edu/~billm/cs265/project/rtl.html) RTL has a Lisp-like external representation, which is why it's noted here.

Original Lisp

"Compilation of Functional Programming Languages using GCC - Tail Calls" by Andreas Bauer (http://home.in.tum.de/~baueran/thesis/baueran_thesis.pdf), has some RTL examples.

Section 2.3 shows a trivial C program:

```
int foo ()
{
   return bar (5);
}
```

Here is an incomplete translation to RTL expressions (before any sibling call optimization):

```
(call_insn 19 9 20 (nil) (call_placeholder 16 10 0 0
    (call_insn 17 16 18 (nil)
    (set (reg:SI 0 eax)
             (call (mem:QI (symbol_ref:SI ("bar")) [0 S1 A8])
                     (const_int 4 [0x4]))) -1 (nil)
         (expr_list:REG_EH_REGION (const_int 0 [0x0])
              (nil))
         (nil))) -1 (nil)
    (nil)
    (nil))

(insn 20 19 21 (nil) (set (reg:SI 58)
         (reg:SI 59)) -1 (nil)
    (nil))

(jump_insn 21 20 22 (nil) (set (pc)
         (label_ref 25)) -1 (nil)
    (nil))

(barrier 22 21 23)

(note 23 22 27 NOTE_INSN_FUNCTION_END)

(insn 27 23 28 (nil) (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))

(insn 28 27 25 (nil) (clobber (reg:SI 58)) -1 (nil)
    (nil))

(code_label 25 28 26 6 "" [0 uses])

(insn 26 25 29 (nil) (set (reg/i:SI 0 eax)
        (reg:SI 58)) -1 (nil)
    (nil))

(insn 29 26 0 (nil) (use (reg/i:SI 0 eax)) -1 (nil)
    (nil))
```

Infix non-default

Below is a sweet-expression equivalent. One question: Should lists here be represented as x(...) or (x ...)? They are equivalent; it's merely a matter of what is clearer. The first parameter is a special marker, so it's reasonable to represent them as x(...); besides, it helps to show off sweet-expressions. I won't do that with strings; a list beginning with a string will be shown as ("x"). RTL isn't really typical

Lisp, but has its own syntactic extensions. I'll show RTL bracketed expressions [...] exactly as they would be in RTL (leaving them as-is). So here's one approach to representing RTL:

```
call_insn 19 9 20 nil()
  call_placeholder 16 10 0 0
    call_insn 17 16 18 nil()
     set reg:SI(0 eax)
       call(mem:QI(symbol_ref:SI(("bar")) [0 S1 A8]) const_int(4 [0x4]))
     -1
     nil()
     expr_list:REG_EH_REGION const_int(0 [0x0]) nil()
     nil()
  -1
  nil()
  nil()
  nil()

insn 20 19 21 nil() set(reg:SI(58) reg:SI(59)) -1 nil() nil()

jump_insn 21 20 22 nil() (set pc() (label_ref 25)) -1 nil() nil()

barrier 22 21 23

note 23 22 27 NOTE_INSN_FUNCTION_END

insn 27 23 28 nil() clobber(reg/i:SI(0 eax)) -1 nil() nil()

insn 28 27 25 nil() clobber(reg:SI(58)) -1 nil() nil()

code_label 25 28 26 6 "" [0 uses]

insn 26 25 29 nil() set(reg/i:SI(0 eax) reg:SI(58)) -1 nil() nil()

insn 29 26 0 nil() use(reg/i:SI(0 eax)) -1 nil() nil()
```

MELT (MiddleEndLispTranslator)

MELT (MiddleEndLispTranslator) (http://gcc.gnu.org/wiki/MiddleEndLispTranslator) is a "high-level Lisp-like language designed to fit very closely in the [GNU Compile Collection (GCC)] internal representations, thru an automated translation of MELT code into GCC

specific C code, compiled by a C compiler (usually some GCC) and then loaded as plugins. This enables easier development of high-level static analysis and transformation, working on GCC middle end representation (GIMPLE tuple)." If you're interested in the general topic of gcc plug-ins, see "Plugging into GCC" (lwn.net) (http://lwn.net/Articles/301135/).

The MELT work is partly funded by the French Ministery of Economy, thru ITEA within the GlobalGCC (GGCC) project (http://www.ggcc.info/). The lead, Basile Starynkevitch, has contributed his GCC contributions using a copyright transfer signed by CEA to FSF. It was presented at the GCC Summit 2007. A special MELT branch was created on February 19, 2008.

Semantically, MELT is a "Lisp1" Lisp dialect (so it's more like Scheme than like Common Lisp regarding names and bindings). You can define primitives, which get translated to C, compiled, and linked into the compiler during the compilation process. The (unboxed) integer addition is pre-defined in MELT as:

```
  (defprimitive +i (:long a b) :long "((" a ") + (" b "))")
```

Where "the first :long occurrence describes the types of the formal arguments a and b, the second occurrence describes the result. There is an minimal object system (single-inheritance hierarchy, rooted at CLASS_ROOT... Tail-recursion is not handled (looping should use the forever keyword, and loops are can be exited)".

Original Lisp

Here is an example from the main MELT page. This is "a sample MELT function repeat-times [that applies] a function f to an argument x some specified n times. f and x are values, n is an unboxed long argument.":

```
(defun repeat-times (f x :long n) ; n is an unboxed formal argument of type long
  (forever reploop                ; infinite loop called reploop
     (if (<=i n 0)                ; test if the unboxed n is negative
         (exit reploop))          ; exit the loop if yes
     (f x)                        ; call f
     (setq n (-i n 1))))          ; decrement n
```

Infix non-default

Here is a sweet-expression equivalent:

```
defun repeat-times (f x :long n) ; n is an unboxed formal argument of type long
  forever reploop                ; infinite loop called reploop
    if {n <=i 0}                 ; test if the unboxed n is negative
      exit reploop               ; exit the loop if yes
    f x                          ; call f
    setq n -i(n 1)               ; decrement n
```

This is an interesting case regarding "infix default" vs. "infix non-default". The comparison operator here is "<=i" - and as a result, the comparison wouldn't be automatically detected by the proposed automatic infix detection rules. This is another argument for not trying to automatically detect infix operators; too often they will be missed, so don't bother. Instead, let the developer specify them using a single uniform syntax.

Satisfiability Modulo Theories Library (SMT-LIB) 1.2

The Satisfiability Modulo Theories Library (SMT-LIB) (http://www.smt-lib.org/)
The major goal of the SMT-LIB initiative is to "establish a library of benchmarks for Satisfiability Modulo Theories, that is, satisfiability of formulas with respect to background theories for which specialized decision procedures exist... [these] have applications in formal verification, compiler optimization, and scheduling, among others... the initiative first aims at establishing a common standard for the specification of benchmarks and of background theories."

The SMT-LIB syntax is "attribute-based and Lisp-like". It permits a syntactic category "user value", used for user-defined annotations or attributes, that start with an open brace "{" and end with a closed brace "}".

Original Lisp

The QF_RDL benchmarks's "check" subset includes this small benchmark as bignum_rd1.smt. I've re-indented the last lines slightly so that it fits inside the usual 80 columns:

```
(benchmark bignum
  :source { SMT-COMP'06 Organizers }
  :notes "This benchmark is designed to check if the DP supports bignumbers."
  :status sat
    :difficulty { 0 }
    :category { check }
  :logic QF_RDL
  :extrafuns ((x1 Real))
    :extrafuns ((x2 Real))
    :extrafuns ((x3 Real))
    :extrafuns ((x4 Real))
  :formula
  (and (<= (- x1 x2) (/ 1 10000000000000000000000000000000))
       (<= (- x2 x3) (/ 1 20000000000000000000000000000011))
       (<= (- x3 x4) (~ (/ 1 10000000000000000000000000000000)))
       (<= (- x4 x1) (~ (/ 1 20000000000000000000000000000012)))))
```

The indentation above is misleading, since "difficulty" isn't a part of ":status". So let's first use a reasonable indentation, trying to make it as readable as possible without changing Lisp syntax. Also, "user values" are written with {...}, which isn't standard Lisp; let's rewrite them as strings ("...."), since they have essentially the same role (they are uninterpreted data):

```
(benchmark bignum
  :source "SMT-COMP'06 Organizers"
  :notes "This benchmark is designed to check if the DP supports bignumbers."
  :status sat
  :difficulty "0"
  :category "check"
  :logic QF_RDL
  :extrafuns ((x1 Real))
  :extrafuns ((x2 Real))
  :extrafuns ((x3 Real))
  :extrafuns ((x4 Real))
  :formula
  (and (<= (- x1 x2) (/ 1 10000000000000000000000000000000))
       (<= (- x2 x3) (/ 1 20000000000000000000000000000011))
       (<= (- x3 x4) (~ (/ 1 10000000000000000000000000000000)))
       (<= (- x4 x1) (~ (/ 1 20000000000000000000000000000012)))))
```

Infix non-default

One problem with this data is that there is an implied syntax that isn't actually embedded in the Lisp formatting. Namely, an atom beginning with ":" is followed by a parameter in this syntax. A pretty-printer that wasn't specially rigged with this convention would not show this format in a pretty way, and a normal sweet-expression reader wouldn't know about this either. This means that the "obvious" sweet-expression notation isn't right. E.G., this fragment:

```
benchmark bignum
  :source "SMT-COMP'06 Organizers"
  :notes "This benchmark is designed to check if the DP supports bignumbers."
  :status sat
```

Would be interpreted as: (benchmark bignum (:source "SMT-COMP'06 Organizers") (:notes "This benchmark is designed to check if the DP supports bignumbers.") (:status sat))

We *could* use this kind of formatting, but I find it ugly and counter-intuitive; it also wastes lots of space:

```
benchmark bignum
  :source
  "SMT-COMP'06 Organizers"
  :notes
  "This benchmark is designed to check if the DP supports bignumbers."
  :status
  sat
```

But this isn't really a problem. One way to make this "pretty" would be to use "(...)" at an outer level - such as with function-calling syntax - which disables indentation processing. Then, we can choose any indentation we like, just as we can in traditional Lisp syntax. We can still use {...} for infix operators. In our examples, I won't use "and" as an infix operator, because the sub-expressions are so long, but I will use infix for "<=", "-", and "/". I'll also use "~" (unary minus) as a one-parameter function.

```
benchmark( bignum
  :source "SMT-COMP'06 Organizers"
  :notes "This benchmark is designed to check if the DP supports bignumbers."
  :status sat
  :difficulty "0"
  :category "check"
  :logic QF_RDL
  :extrafuns ((x1 Real))
  :extrafuns ((x2 Real))
  :extrafuns ((x3 Real))
  :extrafuns ((x4 Real))
  :formula
    and  { {x1 - x2} <= {1 / 10000000000000000000000000000000} }
         { {x2 - x3} <= {1 / 20000000000000000000000000000011} }
         { {x3 - x4} <= ~({1 / 10000000000000000000000000000000}) }
         { {x4 - x1} <= ~({1 / 20000000000000000000000000000012})} )
```

The above looks reasonable enough. But if you were able to change the required expressions, you could make the structure of the parameters much clearer by connecting the parameter names and their values inside lists, instead of merely implying it through a naming convention. This would be an annoyance in

traditional Lisp, because it'd require additional parentheses for each parameter, but this is a non-problem for sweet-expressions. If this was done, you could write it this way:

```
benchmark bignum
  :source "SMT-COMP'06 Organizers"
  :notes "This benchmark is designed to check if the DP supports bignumbers."
  :status sat
  :difficulty "0"
  :category "check"
  :logic QF_RDL
  :extrafuns ((x1 Real))
  :extrafuns ((x2 Real))
  :extrafuns ((x3 Real))
  :extrafuns ((x4 Real))
  :formula
    and  { {x1 - x2} <= {1 / 10000000000000000000000000000000} }
         { {x2 - x3} <= {1 / 20000000000000000000000000000011} }
         { {x3 - x4} <= ~({1 / 10000000000000000000000000000000}) }
         { {x4 - x1} <= ~({1 / 20000000000000000000000000000012})}
```

One challenge with this format is that there's an implied pairing of list elements that doesn't actually result in list pairing... so the indentation processing doesn't help. A similar problem occurs with Arc's "if".

I posted a splicing proposal on the mailing list (http://www.mail-archive.com/readable-discuss@lists.sourceforge.net/msg00124.html), using the backslash character. When doing indentation processing, if the first character of a form is "\" followed by whitespace:

1. If it's the last character of the line (other than 0 or more spaces/tabs), then the newline is considered a space, and the next line's indentation is irrelevant. This continues the line. (Note that comments cannot follow, because that would be confusing.)
2. If it's between items on a line, it's interpreted as a line break to the same indentation level.

3. Otherwise, if it's at the beginning of a line (after 0+ spaces/tabs), it's ignored - but the first non-whitespace character's indentation level is used.

So Arc's:

```
(if
  (condition1) (dothis1)
  (condition2) (dothis2)
  default-result)
```

Could be written these ways:

```
; When condition1, dothis1, etc. are lengthy, you can do this:
if
  condition1()
  \ dothis1()
  condition2()
  \ dothis2()
  default-result

; When condition1, dothis1, etc. are short, you can do this:
if
  condition1() \ dothis1()
  condition2() \ dothis2()
  default-result
```

So we can now write the SMT-LIB examples this way:

```
benchmark bignum
  :source \ "SMT-COMP'06 Organizers"
  :notes \ "This benchmark is designed to check if the DP supports bignumbers."
  :status \ sat
  :difficulty \ "0"
  :category \ "check"
  :logic \ QF_RDL
  :extrafuns \ ((x1 Real))
  :extrafuns \ ((x2 Real))
  :extrafuns \ ((x3 Real))
  :extrafuns \ ((x4 Real))
  :formula
  \ and  { {x1 - x2} <= {1 / 10000000000000000000000000000000} }
         { {x2 - x3} <= {1 / 20000000000000000000000000000011} }
         { {x3 - x4} <= ~({1 / 10000000000000000000000000000000}) }
         { {x4 - x1} <= ~({1 / 20000000000000000000000000000012})} )

; or alternatively:
benchmark bignum \
  :source "SMT-COMP'06 Organizers" \
  :notes "This benchmark is designed to check if the DP supports bignumbers." \
  :status sat \
  :difficulty "0" \
  :category "check" \
  :logic QF_RDL \
  :extrafuns ((x1 Real)) \
  :extrafuns ((x2 Real)) \
  :extrafuns ((x3 Real)) \
  :extrafuns ((x4 Real)) \
  :formula \
  and  { {x1 - x2} <= {1 / 10000000000000000000000000000000} }
         { {x2 - x3} <= {1 / 20000000000000000000000000000011} }
         { {x3 - x4} <= ~({1 / 10000000000000000000000000000000}) }
         { {x4 - x1} <= ~({1 / 20000000000000000000000000000012})} )

; If you reorder the ":formula" entry you can see why just the "\ at the end"
; doesn't completely solve the problem.  You can't really do that here:
benchmark bignum
  :logic \ QF_RDL
  :formula
  \ and  { {x1 - x2} <= {1 / 10000000000000000000000000000000} }
         { {x2 - x3} <= {1 / 20000000000000000000000000000011} }
         { {x3 - x4} <= ~({1 / 10000000000000000000000000000000}) }
         { {x4 - x1} <= ~({1 / 20000000000000000000000000000012})} )
  :source \ "SMT-COMP'06 Organizers"
  :notes \ "This benchmark is designed to check if the DP supports bignumbers."
  :status \ sat
  :difficulty \ "0"
  :category \ "check"
  :extrafuns \ ((x1 Real))
```

```
  :extrafuns \ ((x2 Real))
  :extrafuns \ ((x3 Real))
  :extrafuns \ ((x4 Real))
```

This has a nice and very convenient side-effect; you can create single-line sequences when they make sense. E.G.:

```
define showstats()
  write a \ write b \ write c

; is the same as:
define showstats()
  write a
  write b
  write c

; Which is the same as:
(define (showstats)
  (write a) (write b) (write c))
```

Thus, "\ " as an in-line separator is a lot like ";" as a statement terminator or separator in ALGOL-descended languages (like C and Pascal). That's useful when you have highly related sets of short statements.

Satisfiability Modulo Theories Library (SMT-LIB) 2.0

The Satisfiability Modulo Theories Library (SMT-LIB) (http://www.smt-lib.org/) was discussed above; version 2.0 had a number of big changes.

Original Lisp

Here's an example from section 3.6 of the SMT-LIB 2.0 specification dated August 28, 2010:

```
(forall ((x (List Int)) (y (List Int)))
   (= (append x y)
      (ite (= x (as nil (List Int)))
             y
             (let ((h (head x)) (t (tail x)))
                (insert h (append t y))))))
```

This is an example of ite(x,y,z), an "if-then-else" that returns y if x is true, otherwise it returns z (this is not included in traditional first order logic, a major weakness in the traditional language). The "ite" operator was added in version 2.0 of SMT-LIB.

Infix is not Default (Version 0.2)

Here is one way to rewrite this in a readable format. In this example I've chosen to use "group" for the lists of variables in forall and let, but given short lists like these you could also use traditional list notation:

```
forall
   group
     x List(Int)
     y List(Int)
   = append(x y)
     ite {x = as(nil List(Int))}
       y
       let
         group
           h head(x)
           t tail(x)
         insert h append(t y)
```

NewLisp

NewLisp (http://www.newlisp.org/) is "Lisp-like, general purpose scripting language", with an implementation released under the GPL.

Original Lisp

Here is a sample from their Code patterns (http://www.newlisp.org/CodePatterns.html) document:

```
    (dolist (file-name (3 (main-args)))
        (set 'file (open file-name "read"))
        (println "file ---> " file-name)
        (while (read-line file)
            (if (find (main-args 2) (current-line) 0)
            (write-line)))
        (close file))
```

newLISP

This is the original formatting. It's a little misleading; the write-line is actually *inside* the "if", not a sibling of it. Yet another example of how formatting can mislead a reader, when humans use it but computers don't.

Here's another example from its documentation on apply:

```
(define (gcd_ a b)
    (let (r (% b a))
        (if (= r 0) a (gcd_ r a))))

(define-macro (my-gcd)
    (apply gcd_ (args) 2))
```

Infix is not Default (Version 0.2)

Same thing for the first one, with sweet-expressions:

```
    dolist (file-name (3 (main-args)))
        set 'file open(file-name "read")
        println "file ---> " file-name
        while read-line(file)
            if find(main-args(2) current-line() 0)
              write-line()
        close file
```

Here's a readable version of the gcd and apply example::

```
define gcd_(a b)
    let r %(b a)
        if {r = 0} a gcd_(r a)

define-macro my-gcd()
    apply gcd_ (args) 2
```

Clojure

Clojure (http://clojure.org/) is a Lisp dialect running on top of a Java JVM. differences with other Lisps (http://clojure.org/lisps) explains some of the differences.

Clojure is inspired by Lisp, but it also has several syntactic additions. As discussed in its section on the reader (http://clojure.org/reader), its syntax includes support for:

- Lists. Lists are zero or more forms enclosed in parentheses: (a b c)
- Vectors. Vectors are zero or more forms enclosed in square brackets: [1 2 3]
- Maps. Maps are zero or more key/value pairs enclosed in braces: {:a 1 :b 2} Commas are considered whitespace, and can be used to organize the pairs: {:a 1, :b 2} Keys and values can be any forms.
- Sets. Sets are zero or more forms enclosed in braces preceded by #: #{:a :b :c}

Original Lisp

The example of Clojure agents (http://clojure.org/agents) gives this example, which is "an implementation of the send-a-message-around-a-ring test. A chain of n agents is created, then a sequence of m actions are dispatched to the head of the chain and relayed through it":

```
(defn setup [n next]
  (if (zero? n)
     next
   (recur (dec n) (agent {:next next}))))
(defn relay [x m]
  (when (:next x)
    (send (:next x) relay m))
  (when (and (zero? m) (:report-queue x))
    (. (:report-queue x) (put m)))
  x)
(defn run [m n]
 (let [q (new java.util.concurrent.SynchronousQueue)
       tl (agent {:report-queue q})
       hd (setup (dec n) tl)]
   (doseq m (reverse (range m))
      (send hd relay m))
   (. q (take))))
; Time 1 million message sends:
(time (run 1000 1000))
```

Tim Bray loves Clojure, but not its syntax (http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses), and gives this example:

```
(apply merge-with +
  (pmap count-lines
    (partition-all *batch-size*
      (line-seq (reader filename)))))
```

Infix is not Default (Version 0.2)

Sweet-expressions work well with Clojure, but a few notes should be made first. Clojure uses [...] to notate vectors; to me, this suggests that the rule for sweet-expressions should be 'interpret unprefixed [...] as whatever the base language does' (which would help Clojure and Arc, among others). For sweet-expressions, I'll assume that [...] disable indentation processing inside, just as (...) and {...} do. In Scheme, unprefixed [...] should be considered the same as (...), since that's the meaning for Scheme R6. So, Arc and Clojure have refined the sweet-expression rules for [...].

It's easy to imagine an extension of sweet-expressions that has additional syntactic support for special types, just as Clojure's built-in syntax does. But for the moment, I'll just use map(...) to transform a list into a map, instead of having more syntactic support. Clojure uses curly braces for maps, but we'll continue to use curly braces for infix lists.

Finally, one odd thing is that in Closure, "." is a symbol, and an important one you can include at the beginning of a list. That is weird; in many Lisps, "(. hi)" is the same as "hi" because of the way list reading is typically implemented, and I presumed that when I spec'ed sweet-expressions. For the moment, I'll write the symbol "." as "\.".

Given all that, here's one way to notate this in sweet-expressions:

```
defn setup [n next]
  if zero?(n)
    next
    recur dec(n) agent(map(:next next))
defn relay [x m]
  when :next(x)
    send :next(x) relay m
  when {zero?(m) and :report-queue(x)}
    \. :report-queue(x) put(m)
  x
defn run [m n]
 let [q new(java.util.concurrent.SynchronousQueue)
      tl agent(map(:report-queue q))
      hd setup(dec(n) tl)]
   doseq m reverse(range(m)) send(hd relay m)
   \. q take()
; Time 1 million message sends:
time(run(1000 1000))
```

And here is Bray's, rewritten:

```
apply merge-with +
  pmap count-lines
    partition-all *batch-size*
      line-seq reader(filename)
```

ISLisp

ISLisp (http://en.wikipedia.org/wiki/ISLisp) (also capitalized as ISLISP) is a programming language standardized by the ISO. Its intent was define a small core language based on only the features shared between existing LISPs, particularly Common Lisp, EuLisp, Le Lisp, and Scheme. It provides basic functionality and object-orientation, and was intended to "give priority to industrial needs over academic needs". It has separate function and value namespaces (hence it is a Lisp-2). It is standardized as ISO/IEC 13816:1997 and later revised as ISO/IEC 13816:2007 - Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP. ISLisp is rarely used; Common Lisp and Scheme are *far* more common.

Original Lisp

Sadly, ISO still hasn't arrived at the 21st century, so it still doesn't release standards to the web as a matter of course. A draft of the ISLisp spec draft 23 (http://ryujin.kuis.kyoto-u.ac.jp/~yuasa/lispwg/is23.pdf) is available, and it gives these examples for "and":

```
(and (= 2 2) (> 2 1)) ; t
(and (= 2 2) (< 2 1)) ; nil
(and (eql a a) (not (> 1 2))) ; t
(let ((x a)) (and x (setq x b))) ; b
(let ((time 10))
  (if (and (< time 24) (> time 12))
      (- time 12) time)) ; 10
```

Infix is not Default (Version 0.2)

Here's a version using sweet-expressions:

```
{{2 = 2} and {2 > 1}} ; t
{{2 = 2} and {2 < 1}} ; nil
{{a eql a} and not({1 > 2})} ; t
let ((x a)) {x and setq(x b)} ; b
let ((time 10))
  if {{time < 24} and {time > 12}}
     {time - 12} time ; 10
```

COMFY-65

"The COMFY 6502 Compiler" by Henry G. Baker (http://home.pipeline.com/~hbaker1/sigplannotices/sigcol04.pdf) is an an implementation of the COMFY language (http://home.pipeline.com/~hbaker1/sigplannotices/sigcol03.pdf), which is intended to be a replacement for assembly languages when programming on

`bare' machines. COMFY-65 is specifically targetd for the MOS 6502 8-bit processor, which, as the brains of the Apple II and the Atari personal computers, was one of the most popular microprocessors of all time. Its author describes COMFY-65 as follows:

> "COMFY-65 is a `medium level' language for programming on the MOS Technology 6502 microcomputer [MOSTech76]. COMFY-65 is `higher level' than assembly language because 1) the language is structured- while-do, if-then-else, and other constructs are used instead of goto's; and 2) complete subroutine calling conventions are provided, including formal parameters. On the other hand, COMFY-65 is `lower level' than usual compiler languages because there is no attempt to shield the user from the primitive structure of the 6502 and its shortcomings. Since COMFY-65 ismeant to be a replacement for assembly language, it attempts to provide for the maximumflexibility; in particular, almost every sequence of instructions which can be generated by an assembler can also be generated by COMFY. This flexibility is due to the fact that COMFY provides all the non-branching operations of the 6502 as primitives. Why choose COMFY over assembly language? COMFY provides most of the features of assembly language with few of the drawbacks...

COMFY is really nothing like traditional LISP, so a little explanation is needed here. It's compiling to the 6502 8-bit processor, which is very limited, and it has interesting semantics:

> "Executable instructions in COMFY come in three flavors: tests, actions, and jumps. Tests have two possible outcomes: succeed and fail and therefore have two possible continuations-i.e., streams of instructions to execute next. If the test succeeds, the win continuation is executed; if the test fails, the lose continuation is executed. On the 6502, the tests are carry, zero, negative, and overflow, which succeed if the corresponding flags are on and fail if they are off. Actions are simply executed and always succeed; therefore the win continuation always follows and the lose continuation is always ignored. On the 6502, the actions are all the instructions which do not divert the program counter. Jumps are executed and ignore both their continuations. On the 6502 the only two jump instructions are Return (from subroutine) and Resume (after interrupt)."

COMFY compiles branches extremely efficiently, and it eliminates the need to create lots of names just to give branches to go to. It also includes a very sophisticated macro capability (as you'd expect for a language using Lisp constructs).

Some key functions:

- **(not e)**
  not is a unary operator which has a COMFY expression as an argument. not has the effect of interchanging the win and lose continuations for its argument expression. In other words, the win continuation of (not e) becomes the lose continuation of e and the lose continuation of (not e) becomes the win continuation for e.
- **(seq e1 e2 ... en)**
  seq takes a sequence of COMFY expressions and tries to execute them in sequence. If they all succeed, then the whole expression succeeds. If any one

fails, the sequence is immediately terminated and the lose continuation for the whole expression is executed.

- **(if e1 e2 e3)**
  if takes as arguments three expressions-e1, e2, and e3. COMFY first executes e1 and if it succeeds, e2 is executed. The success or failure of e2 then determines the success or failure of the whole if expression. If, on the other hand, e1 fails, then e3 is executed and its success or failure determines that for the whole if expression. In other words, if uses the success or failure of e1 to choose which of e2 or e3 to execute next; whichever one is not chosen is not executed at all. Notice that the failure of e1 cannot cause the failure of the whole expression.
- **(alt e1 e2 ... en)**
  alt is the `dual' of seq. alt takes a sequence of COMFY expressions and tries to execute them in sequence. If they all fail, then the entire alt expression fails. If any one succeeds, the sequence is immediately terminated (i.e., the rest of the sequence is not executed) and the entire alt expression succeeds. In usual usage, the ei are tests; thus, (alt e1 e2) succeeds if and only if either e1 or e2 succeeds (we don't even find out if both would have succeeded because only the first is executed in this case).
- **(compile e win lose)**
  ;;; compile expression e with success continuation "win" and ;;; failure continuation "lose". ;;; "win" an "lose" are both addresses of stuff higher in memory.
- **(fori from to body)**
  Using the 6502 X register ("i" in this language), loop from "from" to "to" executing "body".
- **(l i location)**
  Load from location+i (6502 X register).
- **(st i location)**
  Store into location+i (6502 X register).

- **(NUMBER action)**
  Perform action NUMBER of times; this is a repeat operator.

The symbols are set using setq, which alternates between a name and the values given the name.

Original Lisp

This is an example of COMFY-65 programming, to compute a Universal Product Code (`UPC') parity check digit. "This example is not intended as a tutorial on computing this function, nor as an example of particularly good code, but only to show the flavor of COMFY programming."

```
;;; Universal Product Code Wand parity check.
(setq
 upctable (compile '(seq 13 25 19 61 35 49 47 59 55 11) 0 0)
 code 10           ; upc code buffer
 digit (+ code 12) ; digit buffer
 temp (+ digit 12) ; temporary location.
 upcwand
 (compile
  '(alt
     (seq (fori (\# 6) (\# 12) ; complement right 6 upc digits.
                (l i code)
                (lxor \# 127)
                (st i code))
          (fori (\# 0) (\# 12) ; map codes using upctable.
                (l i code)
                (not
                  (forj (\# 0) (\# 10)
                                (c j upctable)
                               -=\?))           ; fail if equal.
                (stj i digit)) ; store index of upctable.
          decimal                ; set decimal arithmetic mode.
          (l \# 0)               ; clear ac.
          (fori (\# 0) (\# 12)   ; add up the even digits.
             (+ i digit)         ; loop control clears carry!
             i+1)                ; only every other one.
          (st temp)              ; save partial sum.
          c=0                    ; clear the carry.
          (2 (+ temp))           ; multiply by 3.
          (fori (\# 1) (\# 12)   ; add up the odd digits.
                (+ i digit)      ; loop cotrol clears carry.
                i+1)             ; only every other one.
          (lxor \# 15)           ; select low decimal digit.
          =0\?                   ; fails if non-zero.
          return)
     (seq break ; signal failure.
          return))
  0 0))
```

Infix is not Default (Version 0.2)